UCRL-53828

A FAST BOTTOM-UP ALGORITHM FOR COMPUTING THE CUT SETS OF NONCOHERENT FAULT TREES

G. C. Corynen

Lawrence Livermore National Laboratory

Livermore, California, U.S.A.

November 1987

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

UCRL-53828
Distribution Category UC-32

A FAST BOTTOM-UP ALGORITHM FOR COMPUTING THE CUT SETS OF NONCOHERENT FAULT TREES

G. C. Corynen
Lawrence Livermore National Laboratory
Livermore, California, U.S.A.

November 1987

Preface

The work discussed in this paper is one of the products of several years of basic research to improve the computational efficiency of methods designed to evaluate the reliability, safety, or vulnerability of complex systems.

When this research was initiated, state-of-the-art methods could only produce vague or "soft" estimates for the statistical performance of large discrete systems. In high-risk situations, such estimates lead to unacceptable decision errors, and more accurate and definitive procedures were required. Unfortunately, the quest for greater accuracy and confidence was accompanied with an enormous growth in computational requirements, and this placed most complex systems beyond the reach of methods current at that time.

Given that reliability, safety, and vulnerability are usually expressed as a probability, our first concern was to develop improved probability computation algorithms. This gave rise to the $\Sigma\Pi$ method [1] which, for fault trees, accurately and efficiently computes the probability of the top event from the tree cut sets and their probabilities. Unlike the Inclusion/Exclusion method or other methods currently available, $\Sigma\Pi$ produces absolute upper *and* lower bounds in computation time which is polynomial in the number of cut sets and basic events. As the allocated computer time is increased, these bounds approach each other to any desired accuracy.

But the problem of finding the cut sets is itself computationally complex and, again, existing methods were inadequate to perform this step for systems consisting of more than 100 components.

In this report, we present a new and efficient algorithm, called SHORTCUT, for accomplishing this cut set computation process. Designed to address both coherent and noncoherent fault trees with an arbitrary proportion of common-cause events, it consists of several modules, each of which is discussed in detail in this report.

Whereas the development of $\Sigma\Pi$ and its subalgorithms was supported by the Lawrence Livermore National Laboratory (LLNL), the SHORTCUT work discussed in this report was wholly supported by the Nuclear Regulatory Commission, and was directed by Dr. Dale M. Rasmussen, of the Division of Research.

CONTENTS

List of Figures

Page

A FAST BOTTOM-UP ALGORITHM FOR COMPUTING THE CUT SETS OF NONCOHERENT FAULT TREES

ABSTRACT

An efficient procedure for finding the cut sets of large fault trees has been developed. Designed to address coherent or noncoherent systems, dependent events, shared or common-cause events, the method—called SHORTCUT—is based on a fast algorithm for transforming a noncoherent tree into a quasi-coherent tree (COHERE), and on a new algorithm for reducing cut sets (SUBSET). To assure sufficient clarity and precision, the procedure is discussed in the language of simple sets, which is also developed in this report. Although the new method has not yet been fully implemented on the computer, we report theoretical worst-case estimates of its computational complexity.

Chapter 1

INTRODUCTION

The evaluation of overall system vulnerability or reliability is typically done in two major steps: finding the system failure states (cut sets), and computing the probability of system failure (vulnerability), as shown in Figure 1.

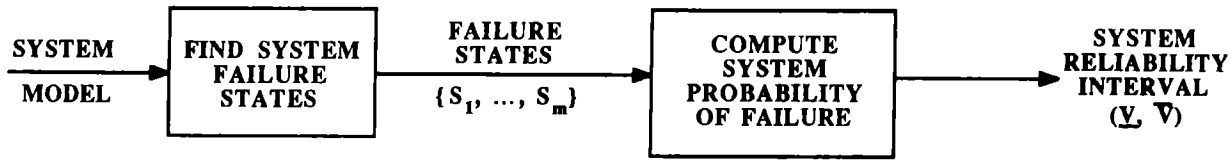


Figure 1. Two principal operations in evaluating system vulnerability or reliability.

A *failure state* is a combination of component or subsystem states which is sufficient—but not necessary—for system failure. When fault trees are used, each component or subsystem is assumed to be in only one of two states: failed or working. Failure states are also called *cut sets*, or *minimal cut sets* if no further Boolean reductions are possible.

Consider thus the family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of minimal cut sets of a fault tree. Then the probability of system failure is

$$V = P(S_1 \text{ or } S_2, \ldots, \text{ or } S_m) \quad , \tag{1}$$

the probability that the system is either in state S_1, or in state S_2, ..., or in state S_m. Mathematically,

$$V = P\left(\bigcup_{i=1}^{m} S_i\right) \quad , \tag{2}$$

the probability of the *union* of all the system cut sets. Each cut set S_i is a combination (*conjunction*) of component states which may be set-theoretically expressed as an inter-

section $S_i = \cap_{j=1}^{n_i} S_{ij}$, where each S_{ij} represents the state of component j in the system failure state i. Therefore,

$$V = P\left(\bigcup_{i=1}^{m} \bigcap_{j=1}^{n_i} S_{ij}\right) \quad . \tag{3}$$

Unfortunately, all the minimal cut sets of a tree can rarely be computed in practice (there are potentially $2^{n/2}$ cut sets for a system with n components), and only a subcollection $\mathcal{S}' = \{S'_1, \ldots, S'_{m'}\} \subset \mathcal{S}$ can usually be found. Only a *lower* bound $\underline{V} = P\left(\cup_{i=1}^{m'} S'_i\right) \leq V$ for V can thus be computed. But if we consider the *dual* fault tree [2], which is simply the original tree with its basic events complemented and its OR gates and AND gates replaced by AND gates and OR gates, respectively, then a lower bound $\underline{V}^D$ for the dual tree can be found in a similar manner. But the top event of the dual tree is the complement of that of the original tree, thus $V^D = 1 - V$, $1 - \underline{V}^D \geq V$, and

$$\underline{V} = P\left(\bigcup_{i=1}^{m'} (S'_i)\right) \leq V \leq 1 - P\left(\bigcup_{i=1}^{m''} (S_i^D)'\right) = \overline{V} \quad , \tag{4}$$

where $\{(S_i^D)' : \ i = 1, \ldots, m''\} = (\mathcal{S}^D)'$, a subcollection of minimal cut sets of the dual tree.

We have thus obtained an interval $[\underline{V}, \overline{V}]$ which is *known* to contain the correct value of system vulnerability. The choice of $\mathcal{S}'$ and $(\mathcal{S}^D)'$ is determined by iterating between Step I and Step II in Figure 1 until a sufficiently tight interval for V is found, or until allocated computer resources are exceeded.

In a previous report [1], a fast method for computing the second operation (Equations 2-4) was presented. This method is particularly useful when very large trees are evaluated whose cut sets do not include many singletons or doubletons, because standard methods such as the Min Cut Upper Bound [3] are inaccurate and unpredictable for such trees, particularly if some basic events have a high probability of occurrence.

The present report is confined to the first operation of Figure 1, and we discuss a new and fast method for finding the cut sets of large noncoherent fault trees with an arbitrary proportion of shared or common-cause basic events. The report is basically structured as

follows.

We start with some mathematical preliminaries in Chapter 2, where we introduce 'simple sets' and present some simple facts about such sets. We also formally define fault trees in that chapter, and we discuss leveled and noncoherent fault trees, and their quasi-coherent versions.

In Chapter 3, we present computational methods for producing leveled, quasi-coherent versions of fault trees. We also discuss the SUBSET algorithm, which reduces a collection of simple sets to a minimal ('reduced') collection, and the TRUNC algorithm, which truncates simple sets in accordance with given rules.

Finally, we present the overall flowchart of the SHORTCUT algorithm, and we report a theoretical estimate of its computational complexity.

Chapter 2

MATHEMATICAL PRELIMINARIES

We first review some basic mathematical concepts associated with simple sets. Then we develop the definitions required for a formal discussion of fault trees.

2.1 Simple Sets

Our methods are based on the concept of a *simple set* [1], and we now review the definitions and propositions required for our discussion.

Consider a set $\mathcal{A} = \{0, 1, -, \emptyset\}$ of four Boolean symbols (the alphabet) whose meanings are defined as follows:

0: False
1: True
−: Don't care
∅: Empty symbol.

Consider next a string $S^n = (e_1, \ldots, e_j, \ldots, e_n)$, $e_j \in \mathcal{A}$, $j = 1, \ldots, n$. Then S^n may be used to denote the product subset $\times_{j=1}^{n} A_j \subset \mathcal{A}^n$ where

$$\begin{array}{lll} A_j = \{0\} & , \text{ if } e_j = 0 & , \\ = \{1\} & , \text{ if } e_j = 1 & , \\ = \emptyset & , \text{ if } e_j = \emptyset & , \\ = \{0, 1\} & , \text{ if } e_j = - & . \end{array} \tag{5}$$

We define such sets S^n as *simple sets of dimension n*. The set of all such sets is $\mathcal{S}^n = \{S^n = (e_i, \ldots, e_j, \ldots, e_n)\colon\ e_j \in \mathcal{A},\ j = 1, \ldots, n\}$, and the simple set $\Omega^n = (-, \ldots, -_n)$ is the *universal set* (every simple set is a subset of this set).

The symbols e_j of a simple set are called the *entries* or *coordinate values* of the simple set.

An *atomic simple set* (an *atom*) is a simple set $A^n = (e_1, \ldots e_j, \ldots, e_n)$, where $e_j \in \{0, 1\} \subset \mathcal{A}$, $j = 1, \ldots, n$. Atoms are thus singletons consisting of a single binary string and constitute the indivisible points from which simple sets are constructed. Simple sets are thus sets in the ordinary sense, whose points are binary strings or atoms.

Simple sets may also be viewed as *product subsets* of $\{0, 1\}^n$, the set of all Boolean strings of length n. For convenience, and unless the dimension n of sets is of particular importance, we shall drop the superscript n from future discussions.

A simple set is *empty* (or *null*) if at least one of its entries e_j equals the null symbol $\emptyset$. Mathematically,

$$S^n = \emptyset \text{ iff } \exists j \ni e_j = \emptyset \quad . \tag{6}$$

An *elementary simple set* is a nonempty simple set which only has one non-don't care coordinate. Elementary simple sets are thus of the form

$$E^n = (-, -, \ldots, e_j, \ldots, -, -, \ldots, -_n) \quad , \tag{7}$$

where $e_j \in \{0, 1\}$ for some single j.

The *projection of a simple set S^n onto the jth coordinate* is the elementary simple set

$$E_j^{S^n} = (-, -, \ldots, e_j^{S^n}, \ldots, -_n) \quad , \tag{8}$$

where $e_j^{S^n}$ is the jth coordinate value of S^n.

Observe that simple sets are a natural representation for Boolean expressions of literals. If we have literals or basic events $E_1, E_j, \ldots, E_n$, for instance, and if we denote the logical AND, OR, and NOT operations by $\wedge$, $\vee$, and $\rceil$, respectively, the conjunction $E_2 \wedge E_{17} \wedge \overline{E}_{19}$ would be represented as the simple set $S = (-, 1_2, \ldots, 1_{17}, \ldots, 0_{19}, \ldots, -_n)$. Disjunctions and complementations (NOTs) will be discussed later.

2.1.1 SOME SIMPLE BUT IMPORTANT FACTS ABOUT SIMPLE SETS

Consider three simple sets: $A = (e_1^A, \ldots, e_j^A, \ldots, e_n^A)$, $B = (e_1^B, \ldots, e_j^B, \ldots, e_n^B)$, and $C = (e_1^C, \ldots, e_j^C, \ldots, e_n^C)$, $e_j^A \in \mathcal{A}$, $e_j^B \in \mathcal{A}$, $e_j^C \in \mathcal{A}$. Let $J = \{j\colon\ j = 1, \ldots, n\}$, the set of all coordinates.

2.1.1.1 Dominations

DEFINITION:

B dominates A in coordinate j, written B dom$(j)A$, iff

$$(e_j^B = -) \bigvee (e_j^A = \emptyset) \bigvee (e_j^B = e_j^A) \quad . \tag{9}$$

Informally, this condition may be paraphrased by the statement, "B dominates A in coordinate j if, and only if, the jth coordinate of B is a don't care, OR the jth coordinate of A in the empty symbol, OR the jth coordinates of A and B are equal."

Similarly, *B strictly dominates A in coordinate j*, written B DOM$(j)A$, iff

$$\left(e_j^A = \emptyset \bigwedge e_j^B \neq \emptyset\right) \bigvee \left(e_j^A \neq - \bigwedge e_j^B = -\right) \quad . \tag{10}$$

The sets of dominating coordinates of B over A are defined as the sets dom$(B/A) = \{j\colon\ B$ dom$(j)A\}$, for ordinary domination, and DOM$(B/A) = \{j\colon\ B$ DOM$(j)A\}$, for strict domination.

2.1.1.2 Complements

The complement of a coordinate entry e_j is denoted by $\overline{e}_j$. Clearly, $\overline{e}_j = 1$ iff $e_j = 0$, and $\overline{e}_j = -$ iff $e_j = \emptyset$.

For elementary simple sets (hence, for projections) $E = (-, \ldots, e_j, \ldots, -)$, the complement $\overline{E}$ of E is $\overline{E} = (-, \ldots, \overline{e}_j, \ldots, -)$, the elementary simple set with its non-don't care coordinate equal to the complement of that of E. For projections E_j^A,

$$\overline{E}_j^A = (-, -, \overline{e}_j^A, \ldots,) \quad . \tag{11}$$

Complements of more general sets are discussed in Section 2.1.1.7.

2.1.1.3 Equalities

Simple sets are equal iff all their coordinates are equal or if they both have one empty symbol in at least one coordinate, in which case they are both empty. Formally,

$$A = B \text{ iff } \left(A = \emptyset \bigwedge B = \emptyset\right) \bigvee \left(\left(e_j^A = e_j^B\right) \quad , \qquad j = 1, \ldots, n\right) \tag{12}$$

2.1.1.4 Intersections

The intersection of two simple sets A and B is the simple set $C = (e_1^C, \ldots, e_j^C, \ldots, e_n^C)$, where $e_j^C = \min\{e_j^A, e_j^B\}$, and

$$\begin{aligned} \min\{\emptyset, \emptyset\} = \min\{\emptyset, e_j\} = \min\{e_j, \emptyset\} = \min\{0, 1\} = \min\{1, 0\} = \emptyset \\ \min\{0, -\} = \min\{-, 0\} = \min\{0, 0\} = 0 \\ \min\{1, -\} = \min\{-, 1\} = \min\{1, 1\} = 1, \text{ and } \min\{-, -\} = - \quad . \end{aligned} \tag{13}$$

Observe that the intersection operation is performed *coordinatewise*, by computing the minimum of the two coordinate values, one coordinate at a time. This convenient formulation does not hold in the computation of unions, differences, and complements, as we show in the following subsections. Note also, that any simple set A is equal to the intersection of all its n projections E_j^A. Thus

$$A = \bigcap_{j=1}^{n} E_j^A \equiv \bigcap\{E_j^A\colon\ j = 1, \ldots, n\} \quad ,$$

obviously,

$$B \bigcap A = \emptyset \quad \text{iff} (\exists j \in J) \ni \left(e_j^A = \emptyset \bigvee e_j^B = \emptyset \bigvee e_j^A = \overline{e}_j^B\right) \quad . \tag{14}$$

2.1.1.5 Unions

In contrast to intersections, there is no succinct componentwise representation of the union $\cup_{i=1}^{m} A_i$ of m simple sets A_i (and the disjunction of expressions which it represents). In general, the union of two or more simple sets cannot be represented as a single simple set. To see this, consider the simple example of two sets $A = (0, 1, 1)$ and $B = (1, 0, -)$. Then A is an atom, and B has two atoms, $B_1 = (1, 0, 1)$ and $B_2 = (1, 0, 0)$. While $B_1 \cup B_2$ is thus a simple set (B), $A \cup B$ cannot be represented as a simple set. This occurs because sets B_1 and B_2 are the same except in one coordinate, where they complement each other, and this coordinate turns into a don't care upon unioning. But this is not the case with sets A and B. In general, therefore, there is no coordinatewise operation which allows the representation of a union of sets as a single set. Consequently, we represent $A \cup B$ as $\cup\{A, B\}$. More generally, we shall represent the union of a collection $\{A_i\colon\ i = 1, \ldots, m\}$ of sets as

$$\bigcup_{i=1}^{m} A_i \text{ or } \bigcup\{A_i\colon\ i = 1, \ldots, m\} \quad . \tag{15}$$

2.1.1.6 Subsets

A is a subset of B iff B dominates A in all coordinates. Mathematically,

$$A \subset B \text{ iff } \quad \mathrm{dom}(B/A) = J \quad , \tag{16}$$

and

$$A \underset{\neq}{\subset} B \text{ iff } \quad (\mathrm{dom}(B/A) = J) \bigwedge (\exists j \ni B\ \mathrm{DOM}(j)A) \quad , \tag{17}$$

the latter representing the case where A is a *strict* subset of B.

As a simple example, consider sets $A = (0, 1, 1)$ and $B = (0, -, 1)$. Then $\mathrm{dom}(B/A) = \{1, 2, 3\}$ and $\mathrm{DOM}(B/A) = \{2\}$, and $A \underset{\neq}{\subset} B$. The singleton $(0, 0, 1)$ is in B but not in A.

2.1.1.7 Differences

The difference of B over A is the set of binary strings in B not in A. Thus,

$$B - A = \bigcup \{S_j = B \bigcap \overline{E}_j^A : \ j \in \mathrm{DOM}(B/A)\} \quad . \tag{18}$$

Since this may not be obvious, we give a simple example. Let $A = (1, -, -, 1, -, -, 1)$ and $B = (-, 1, -, -, 1, -, -,)$. Then

$$B - A = \bigcup \{(0, 1, -, -, 1, -, -,) \ , \ (-, 1, -, 0, 1, -, -) \ , \ (-, 1, -, -, 1, -, 0)\} \ ,$$

each set inside the brackets resulting from the strict domination of B over A in a corresponding coordinate ($\mathrm{DOM}(B/A) = \{1, 4, 7\}$). Complements of elementary simple sets were discussed in Section 2.1.1.2. For a general simple set A, the complement $\overline{A}$ of A is simply the difference $\Omega - A$, the set of points in the universe Ω not in A.

2.2 Fault Trees

Fault trees are structures which operate on simple sets. A careful definition of fault trees is required to discuss methods which we have developed to exploit their structure. In our discussion, we use the words "simple set" and "expression" interchangeably.

2.2.1 BOOLEAN GATES

Starting with three Boolean operations + (OR), · (AND), and ¬ (NOT), a *gate* is a triple $g = \langle I, O, B \rangle$, where

> I is the set of *input variables* ("input terminals") of g
> O is the *output variable* ("output terminal") of g
> $B \in \{+, \cdot, \neg\}$ is the *Boolean operation* of g.

Gate variables I and O range over Boolean expressions in Disjunctive Normal Form (DNF), or unions of simple sets, and $O = B(I)$.

When $B = +$, then g is an *OR gate*, and $O = \vee_{v \in I} v$. When $B = \cdot$, g is an AND gate, and $O = \wedge_{v \in I} v$. When $B = \neg$, g is a NOT gate, both I and O are singletons, and $O = \neg I$, the complement of input variable I, which we occasionally denote by $\bar{I}$. In specific instantiations of a gate g, B does not operate abstractly on I, of course, but on the specific expressions or simple sets associated with terminals in I.

2.2.2 GATE RELATIONS AND INTERCONNECTIONS

Consider a collection of

$$\text{OR gates } G^+ = \{g_i^+ = \langle I_i^+, O_i^+, +_i \rangle \ , \quad i = 1, 2, \ldots, \gamma^+\} \quad ,$$

$$\text{AND gates } G^\cdot = \{g_j^\cdot = \langle I_j^\cdot, O_j^\cdot, \cdot_j \rangle \ , \quad j = 1, 2, \ldots, \gamma^\cdot\} \quad ,$$

and

$$\text{NOT gates } G^\neg = \{g_k^\neg = \langle I_k^\neg, O_k^\neg, \neg_k \rangle \ , \quad k = 1, 2, \ldots, \gamma^\neg\} \quad .$$

Let $O^+ = \cup_{i=1}^{\gamma^+} O_i^+$, the set of all OR gate outputs, $I^+ = \cup_{i=1}^{\gamma^+} I_i^+$, the set of all OR gate inputs, and similarly for $O^\cdot$, $I^\cdot$, $O^\neg$, and $I^\neg$, and let $O = O^+ \cup O^\cdot \cup O^\neg$ and $I = I^+ \cup I^\cdot \cup I^\neg$.

A gate g relates its inputs I_g to its outputs O_g and, neglecting the particular structure of the gate operation, this may be denoted by a relation, $\mathcal{R}_g \subset I \times O$. The set G of all gates thus relates inputs to outputs *via* a *gate relation* $\mathcal{R}_G \subset I \times O$, where $(i, o) \in \mathcal{R}_G$ iff $i \in I$ is the input of some gate $g \in G$ whose output is $o \in O$. This relation will be used below when we discuss interconnections in fault trees.

DEFINITION

An *Interconnection structure on* $G = G^+ \cup G^\cdot \cup G^\neg$ is a relation $\mathcal{I} \subset O \cup I \times O \cup I$ such that $(x, y) \in \mathcal{I}$ iff x is connected to y.

If an "interconnection" simply means that terminals are physically connected together, or variables are identified, then $(x, y) \in \mathcal{I}$ iff $x = y$, in which case $(x, y) \in \mathcal{I}$ iff $(y, x) \in \mathcal{I}$. This is exclusively the case in this report. If $(x, y) \in \mathcal{I}$, x is conventionally called the *predecessor of* y and y is called the *successor of* x.

For most systems, such as fault trees, $\mathcal{I}$ must satisfy certain constraints and has additional properties. These are specified in the next section.

2.2.3 CONSTRAINTS ON INTERCONNECTIONS

In general, not every gate output terminal may be connected to every input terminal, particularly to its own inputs. Since $\mathcal{I}$ is a set of *interconnection pairs*, such constraints may be easily expressed as constraint sets which can be subtracted from the set of allowable interconnections. If inverting gates ($\neg$) may not connect to other inverting gates, for instance, $O^{\neg} \times I^{\neg}$ may be removed, and the relation becomes

$$\mathcal{I} \subset O \cup I \times O \cup I - O^{\neg} \times I^{\neg} \quad . \tag{19}$$

Two principal constraints that must be satisfied in fault trees are intended to prevent *gate feedback*, where the output of a gate is connected directly to one of its inputs, and *gate feedforward*, where an input of a gate is connected directly to its output. These constraints, denoted by C_1 and C_2 are easily represented by subtracting from $\mathcal{I}$ the two sets $CFB = \bigcup_{g \in G}\{(x, y)\colon\ x \in O_g, y \in I_g\}$ and $CFF = \cup_{g \in G}\ \{(x, y)\colon\ x \in I_g, y \in O_g\}$, respectively. Another constraint (C_3) is that no gate outputs may be connected together

$$O \times O \bigcap \mathcal{I} = \emptyset \quad . \tag{20}$$

Informally, there does not exist any pair $(O_1, O_2) \in O \times O$ that is also an interconnection pair, *i.e.*, $(O_1, O_2) \in \mathcal{I}$. Another constraint ($C_4$) requires that $\mathcal{I}$, together with the *gate relation* $\mathcal{R}_g$, be acyclic, since trees contain no cycles. Recall that a relation R is *acyclic* if there does not exist a sequence of pairs $((x_1, x_2), \ldots, (x_k, x_1))$ all of which are in R. We define an interconnection $\mathcal{I}$ on G to be *acyclic* if $\mathcal{I} \cup \mathcal{R}_G$ is an acyclic relation. In terms of terminals, $\mathcal{I}$ on G is acyclic if no terminal is reachable from itself, *i.e.*, there does not exist a sequence $((x_1, x_2), \ldots, (x_k, x_{k+1}))$ of pairs $(x_i, x_{i+1}) \in \mathcal{I} \cup \mathcal{R}_G$ such that $x_{k+1} = x_1$. We also require constraint C_5 that R_F leave no input or output variable isolated. Each variable must have at least one successor or one predecessor under R_F and we then say that R_F is *connecting*.

Finally, constraint C_6 requires that there be one and only one variable which has no successor under R_F. (This variable will be called the *top variable* (top event)).

2.2.4 FAULT TREES

Using the machinery developed in the previous sections, we now have a formal framework for a precise discussion of fault trees. We start with some important definitions.

DEFINITION

Consider a finite set G of gates with gate relation $\mathcal{R}_G$, and an interconnection structure $\mathcal{I}$ on G. A ***Fault Tree*** is a system $F = \langle G, \mathcal{R}_F \rangle$, where $\mathcal{R}_F = \mathcal{I} \bigcup \mathcal{R}_G$ and satisfies the interconnection constraints C_1 through C_6. When $\mathcal{R}_F$ satisfies these constraints, it is called a ***Fault Tree Relation.***

The single variable which has no successor under $\mathcal{R}_F$ is called the *top variable* and this variable is the ouput of the *top gate.* All variables which have no predecessors under R_F are called ***basic variables.***

The top variable T of a fault tree F is a Boolean indicator of the overall state of the system represented by F:

$$T = \begin{cases} 1 \ , & \text{if the system is in the functioning state} \\ 0 \ , & \text{if it is in the failed state.} \end{cases}$$

In a fault tree F, basic variables (called *basic events*) represent the components of the system represented by F, and a principal issue is to determine the combinations of binary component states which are sufficient (but not necessary) for system failure. Component states are represented by a *component state vector* $E = (E_1, \ldots, E_j, \ldots, E_n)$, where

$$E_j = \begin{cases} 1 & \text{if component } j \text{ is in the functioning state} \\ 0 & \text{if component } j \text{ is in the failed state.} \end{cases}$$

These are called the *cut sets* of F, and are obtained by solving a special function defined as follows.

DEFINITION

Let $E = (E_1, \ldots, E_j, \ldots, E_n)$ be the state vector for the basic variables of a fault tree F with top variable T. Then a *structure function for* F is a mapping ϕ_F: $\{0, 1\}^n \rightarrow \{0, 1\}$ such that $T = \phi_F(E)$ is valid. In other words, $T = \phi_F(E) = 1$ iff E is a combination of

component states for which the system is functioning, and conversely for $T = \phi_F(E) = 0$ [3]. The set $C_F = \{E\colon\ \phi_F(E) = 0\}$ is the set of *cut sets* of F.

2.2.4.1 Leveled fault trees

An important special case of a fault tree is a *Leveled Fault Tree*, a tree which has been partitioned into levels (sets) of gates of the same type, in accordance with the interconnection structure $\mathcal{I}$. In such trees, the outputs of OR gates may be connected only to AND gate inputs or NOT gate inputs, and AND gate outputs may connect only to OR gate inputs or NOT gate inputs. Furthermore, NOT gate outputs may not connect to NOT gate inputs, but may be connected to AND or OR gate inputs.

Such trees have a hierarchical structure where the inputs of gates at a given level may be connected only to outputs of gates at a lower level, and may feed only gates at a higher level in the hierarchy. To define such trees, we need the notion of *interconnection sets* of a fault tree.

The interconnection structure $\mathcal{I}$ induces a mapping $F^{-1}\colon\ 2^G \rightarrow 2^G$, where 2^G is the set of subsets of G, as follows. If $A_1 \in 2^G$ (*i.e.*, $A_1 \subset G$), $F^{-1}(A_1)$ is the set of gates g whose outputs O_g are connected to an input of at least one gate in A_1 under the interconnection structure $\mathcal{I}$. Thus, $F^{-1}(A_1) = A_2 \in 2^G$, for some $A_2 \in 2^G$. By induction, we can thus define $F^{-1}\cdot F^{-1}(A_1) = F^{-2}(A_1) = F^{-1}(A_2) = A_3 \in 2^G$, and $F^{-(m+1)}(A_1) = A_m \in 2^G$. Since fault trees are acyclic, there exists some minimal number $m = L$ such that $F^{-(m-k)}(A_1) = \emptyset$ for $m \geq k \geq 1$. Starting from the top gate set $\{g_T\} = A_L$, we can thus construct a collection of distinct nonempty sets $S_I = \{A_1, \ldots, A_{\ell_L}, \ldots, A_L\}$ whose union $\cup_{\ell=1}^{L} A_\ell = G$, where $A_\ell = F^{-(L-\ell)}(A_L)$, and L is the minimal number. We call S_I the *interconnection sets of* $F = \langle G, \mathcal{R}_F \rangle$. To capture the notion of "level," we need to find a partition of G whose elements are gates of the same type, and which respects the interconnection structure $\mathcal{I}$ *via* the interconnection mapping F^{-1}.

DEFINITION

Consider a fault tree $F = \langle G, \mathcal{R}_F \rangle$ whose OR, AND, and NOT gates are G^+, $G^\cdot$, and $G^\rceil$, respectively, and whose interconnection sets are $S_I = \{A_1, \ldots, A_\ell, \ldots, A_L\}$. A *leveling of* F is a partition $\mathcal{L} = \{G_\ell\colon\ \ell = 1, \ldots, L\}$ of G whose elements G_ℓ satisfy the

following conditions for all $\ell = 1, \ldots, L$:

$$\begin{aligned}
&\left(G_\ell \subset G^+ \bigcap A_\ell\right) \bigvee \left(G_\ell \subset G^\cdot \bigcap A_\ell\right) \bigvee \left(G_\ell \subset G^\rceil \bigcap A_\ell\right) \\
&\left(G_\ell \subset G^+\right) \Longrightarrow \left(G_{\ell+1} \subset G^\cdot\right) \bigvee \left(G_{\ell+1} \subset G^\rceil\right) \\
&\left(G_\ell \subset G^\cdot\right) \Longrightarrow \left(G_{\ell+1} \subset G^+\right) \bigvee \left(G_{\ell+1} \subset G^\rceil\right) \\
&\left(G_\ell \subset G^\rceil\right) \Longrightarrow \left(G_{\ell+1} \subset G^+\right) \bigvee \left(G_{\ell+1} \subset G^\cdot\right) \quad . \qquad (21)
\end{aligned}$$

We now define what we mean by a *leveled fault tree.*

DEFINITION

Consider a fault tree $F = \langle G, \mathcal{R}_F \rangle$ which has a leveling $\mathcal{L}_F = \{G_\ell\colon\ \ell = 1, \ldots, L\}$. Then the structure $LF = \langle G, \mathcal{R}_F, \mathcal{L}_F \rangle$, consisting of F together with the leveling $\mathcal{L}_F$, is called a *Leveled Fault Tree.*

Consistent with earlier conventions, we define G_1 as the *bottom level* of LF, $G_L = \{g_L\}$ the *top level* of LF, and G_ℓ as the ℓth level of LF. Recall that the inputs of G_1 are called the basic variables of LF and the output of g_L is called the *top variable* of LF.

2.2.5 LEVELING A FAULT TREE

Without restructuring, many fault trees may not have any levelings. Other fault trees may have multiple levelings. Using the previous two definitions, and other concepts introduced in Section 2.2.4, finding a leveling for a tree is usually straightforward, and we do not develop an algorithm for this operation here. We should note, however, that the inability to level a tree arises because several gates of different types may feed a given gate, or the output of a given gate may be connected to the inputs of several gates of different types. In these circumstances, such "multiple output" gates may be split into multiple gates, one gate for each type of gate connected to the output of the original gate. Such restructuring may not lead to a leveled tree with a *minimum* number of gates, but it will produce at least one leveling. Since the computational burden to compute cut sets is only a weak function of the number of gates, no significant computational penalty is incurred. As an example, Figure 2 illustrates a leveled fault tree with five levels. Note that fault trees

need not be leveled for our methods to apply, only that leveled fault trees are generally easier to visualize and process.

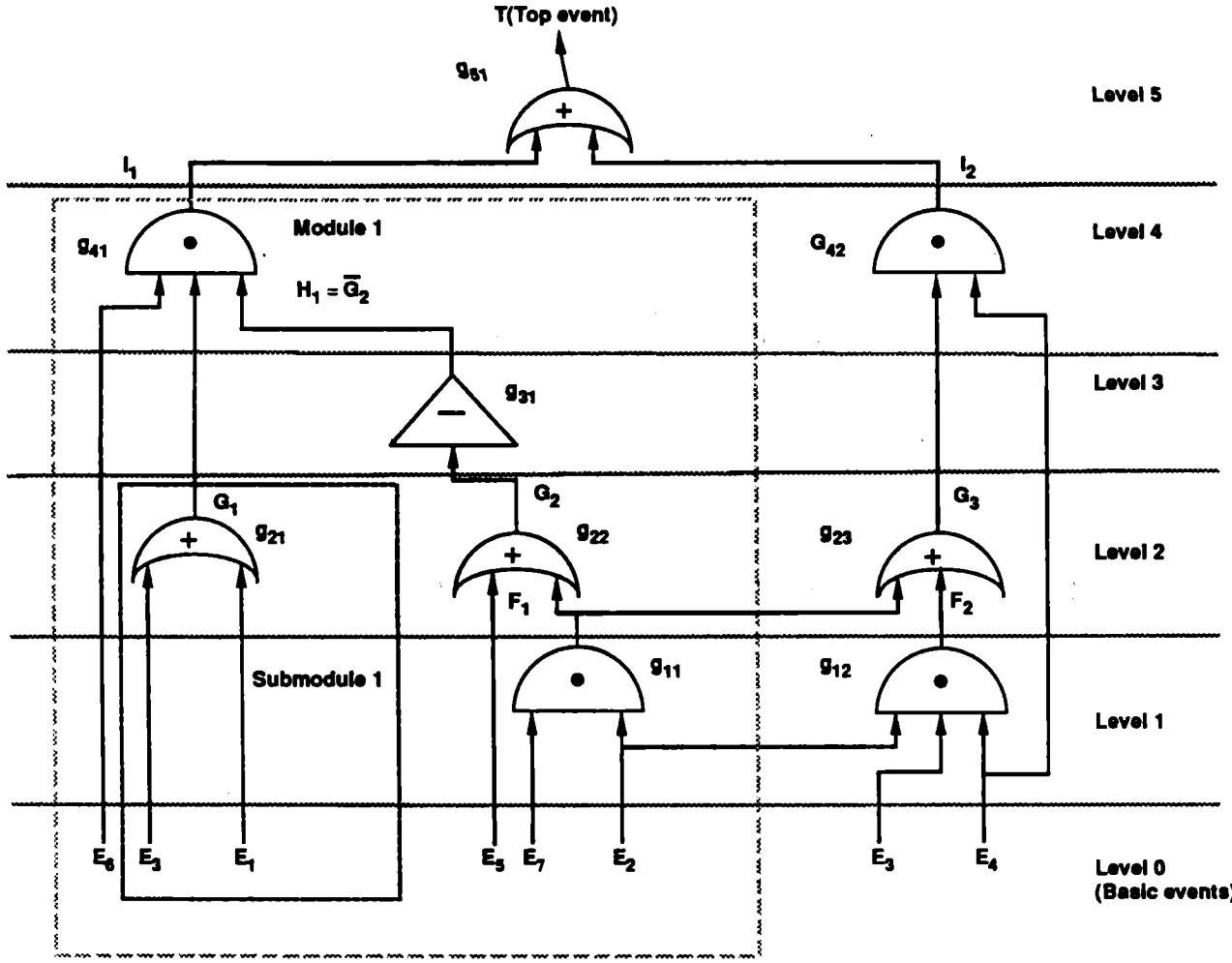


Figure 2. A noncoherent leveled tree with five levels, modules, and submodules.

2.2.6 NONCOHERENT FAULT TREES

A fault tree is *noncoherent* if it is not coherent. To define coherence, we need some preliminary concepts.

DEFINITION

A structure function ϕ is *indifferent* to a component state E_j iff $(\forall E \in \{0,1\}^n)$ $\left(\phi\left(E|_{E_j=1}\right) = \phi\left(E|_{E_j=0}\right)\right)$ where $E|_{E_j=x}$ is the component state vector E with its jth component set to $x \in \{0,1\}$.

DEFINITION

A structure function ϕ is *increasing* iff $(\forall E \in \{0, 1\}^n)\left(\phi\left(E|_{E_j=1}\right) \geq \phi\left(E|_{E_j=0}\right),\ j = 1, 2, \ldots, n\right)$. We now define what we mean by a coherent fault tree.

DEFINITION

A fault tree F with structure function ϕ_F is *coherent* iff

1. ϕ_F is *not indifferent* to any component state E_j

2. ϕ_F is increasing.

Although every noncoherent fault tree must have at least one complemented basic event or must contain at least one NOT gate, or both, the converse is not necessarily true. It is easy to construct a fault tree which has complemented basic events and NOT gates, and yet is coherent. Whether a given fault tree is coherent or not cannot always be determined before computing all its cut sets, except when it contains no NOT gates or complemented basic events. Then it is surely coherent.

Standard methods for computing the cut sets of noncoherent fault trees—or of those suspected of being noncoherent—use de Morgan's Theorem to operate on the cut sets themselves by complementing these as they are derived from the tree [4-8]. But there are potentially $2^{n/2}$ such cut sets, and thus $\mathcal{O}(2^n)$ work may be required to deal with the noncoherent aspects of trees, and this is not practical.

To remove difficulties associated with noncoherence, it suffices to transform a tree into a *quasi-coherent* tree, one that contains only AND gates and OR gates. Such trees may obviously contain complemented basic events and, if none of these events occur in both the complemented and noncomplemented form, the quasi-coherent tree can be further transformed into a fully coherent tree with a simple redefinition of basic variables. In contrast to methods which operate on cut sets, our transformation method operates on the *gates* of a fault tree and may occasionally require that new gates be added to the original tree. As we shall see in the next section, however, the size of the resulting quasi-coherent tree will never exceed twice the size of the original tree. Since the amount of work required to compute cut sets by our method grows very slowly as the number of gates is increased, no significant increase in computational work is required.

2.2.7 QUASI-COHERENT VERSIONS: THE **COHERE** ALGORITHM

The COHERE algorithm reduces noncoherent fault trees to one of their quasi-coherent versions. It is a top-down reduction procedure which uses de Morgan's Theorem [2] iteratively until all NOT operations have been transferred to the bottom level of the tree, where they appear as complements on the basic events. Every NAND combination is replaced by an OR gate with complemented inputs, and every NOR gate combination is replaced by an AND gate with complemented inputs. The NOR combination (g_{22}, g_{31}) of Figure 2, for instance, is replaced by an AND gate with two complemented inputs $\overline{E}_5$ and $\overline{F}_1$.

The process of replacing inverted gates by their duals [2], and of inverting their inputs, continues down the tree until the bottom of the tree is reached.

A slight complication arises when these inverted or complemented outputs are also used as *uncomplemented* inputs to other gates, as F_1 is an input to both g_{22} and g_{23}. Then the reduction process produces two gates, one whose output is the original output, and the dual gate whose output is the complement of the original output; unless the complemented output (event) already appears in the complemented form elsewhere in the tree, in which case the complemented events are simply "connected together," since they are identical.

Referring again to Figure 2, the reduction procedure replaces g_{22} and g_{31} with an AND gate, one of whose inputs is $\overline{F}_1$. But F_1 is an input to g_{23}. Thus, the reduced tree must generate *both* $\overline{F}_1$ and F_1 and this can be done only if both g_{11} *and* its dual g_{11}^D are retained, as shown in the reduced tree of Figure 3.

At first sight, this procedure might appear to produce an explosive growth of gates, but it does not. In the worst case, the quasi-coherent version has at most twice as many gates as the original tree since each inversion creates at most two gates at the next lower level. Observe that, when a shared event requires complementation by several of its sharing gates (event F_1 is shared by two gates g_{22} and g_{23}, for instance) only one new gate must be added since the event is now available in both forms. If gate g_{23} in Figure 2 were also preceded by an inverting gate, for instance, no new gate would be required by g_{23} since $\overline{F}_1$ would already be available from g_{11}^D.

Because the effort required to compute cut sets with our method is only a weak polynomial function of the number of gates in the fault tree, doubling the number of gates

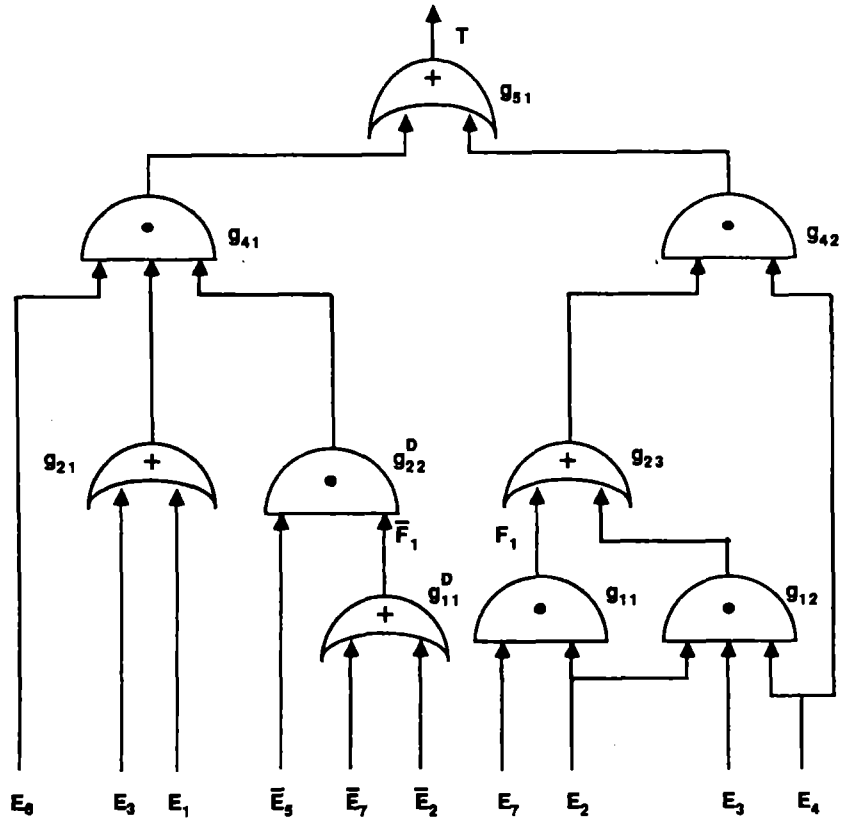
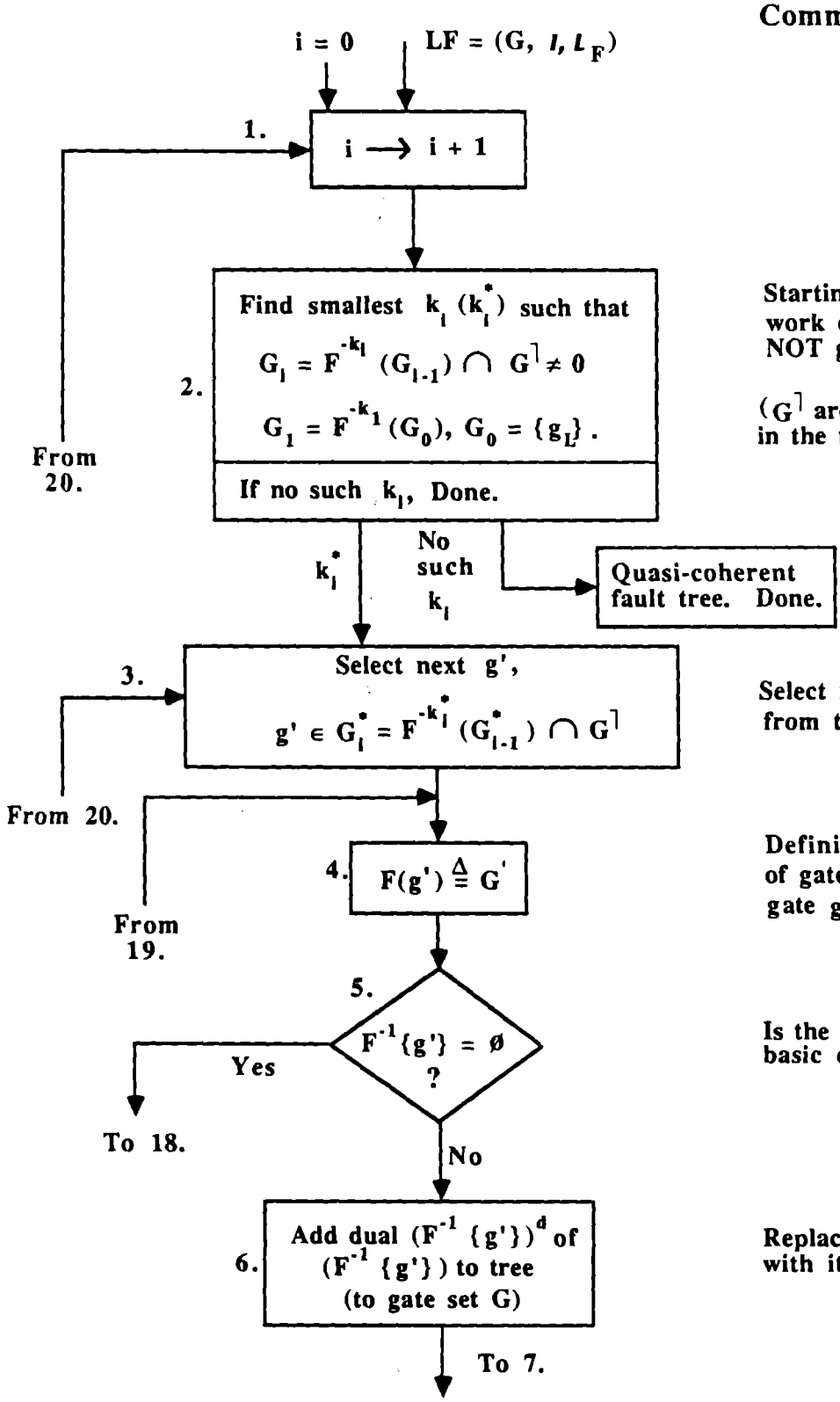


Figure 3. Quasi-coherent version of Figure 2.

does not result in significant increase in work, and the computational advantages of this method over other methods [4,5,9] are significant.

Another significant computational advantage of COHERE will become apparent when it is combined with ΣΠ [1], a method for computing the overall system reliability using disjointing procedures. Cut sets which include complemented basic events are often disjoint from other cut sets, and these require no further processing by subsequent disjointing procedures, such as ΣΠ, thereby reducing even further the overall work required.

We conclude this section with a flowchart of the COHERE algorithm (Figure 4). Recall that, for any set A of gates, $F^{-1}(A)$ is the set of gates each of whose output connects to an input of at least one gate in A, and similarly for the forward relation F. Note that $F \cdot F^{-1}\{g\} \neq \{g\}$ necessarily.



Comments

Starting with top gate g_L, work down the tree until next NOT gates are encountered.

($G^{\neg}$ are all the NOT gates in the tree).

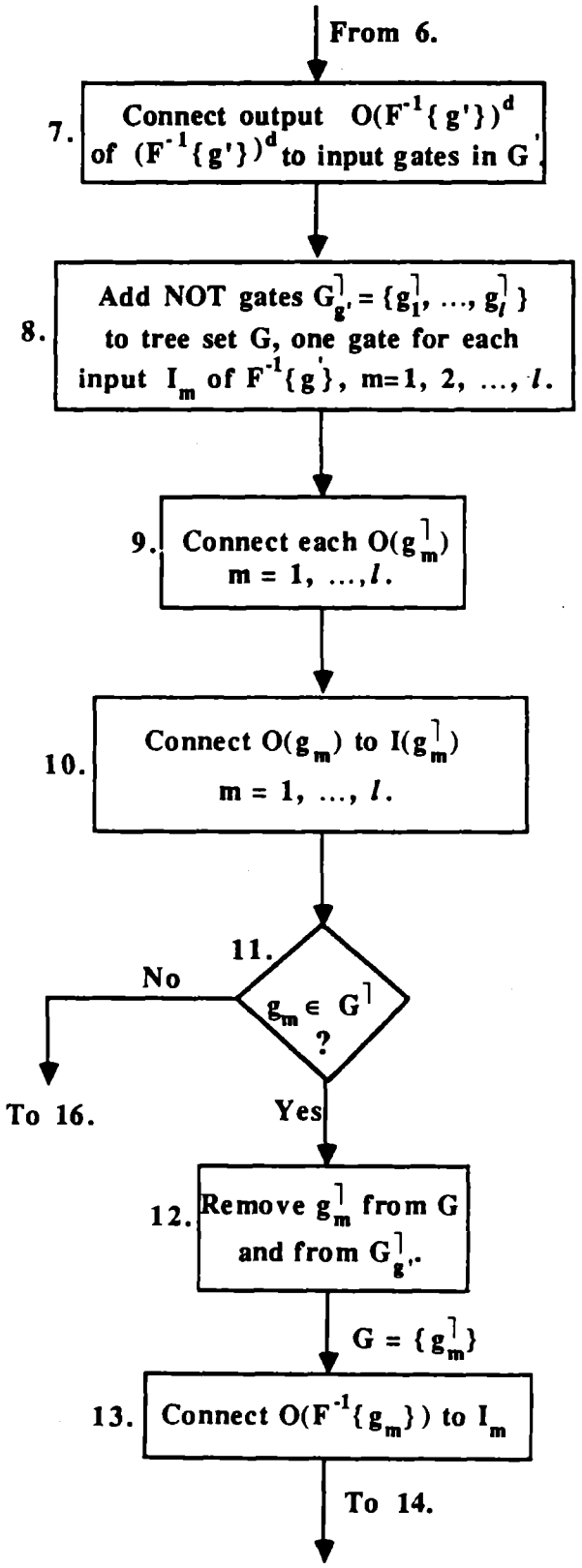
Select next NOT gate g' from the list.

Definition. G' is the set of gates fed by NOT gate g'

Is the input to g' a basic event?

Replace OR or AND with its dual.

Figure 4 (continued)



Start interconnection process for new gate.

Define input set of $(F^{-1}\{g'\})^d$ as $I(F^{-1}\{g'\})^d = \{I_1, \ldots, I_l\}$.

Connect each NOT gate output to a corresponding input I_m of $(F^{-1}\{g'\})^d$.

Let $F^{-2}(\{g'\}) = \{g_1, \ldots, g_l\} \neq \emptyset$. Connect gate outputs to corresponding NOT gate inputs $I(g_m^{\neg})$, m = 1, ..., l.

If any $g_m \in F^{-2}(\{g'\})$ are NOT gates, remove corresponding $g_m^{\neg}$ from tree set G and from $G_g^{\neg}$ list.

Bypass g_m to I_m

Figure 4 (continued)

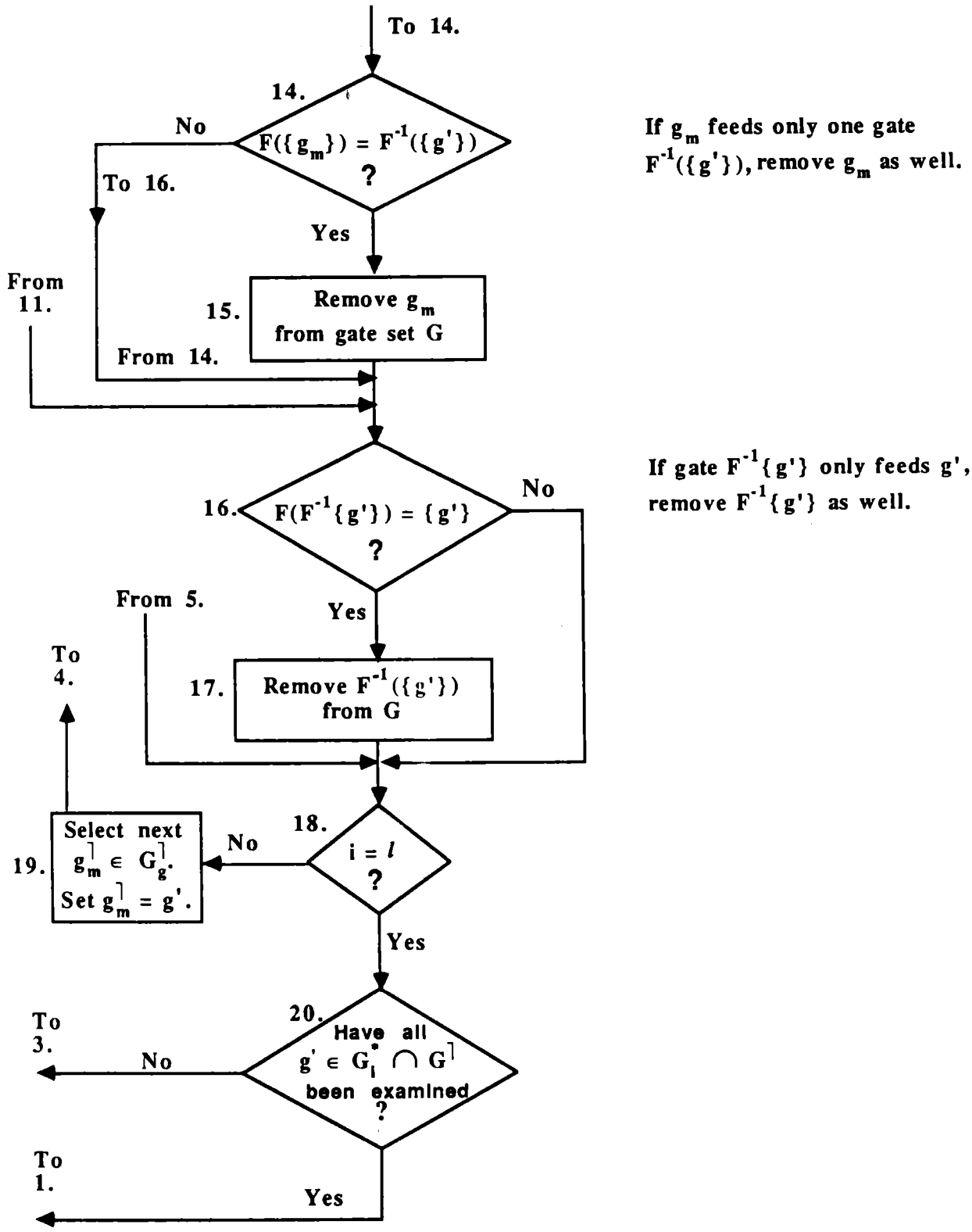


Figure 4. Flow chart of the COHERE algorithm.

Chapter 3

COMPUTING THE CUT SETS OF FAULT TREES:

The SHORTCUT Algorithm

3.1 Introduction

Even when they are very reliable, complex systems usually have an enormous number of failure states. Simple two-state n-component systems such as standard fault trees, for instance, may have up to $2^{n/2}$ failure states. Fortunately, most of these failure states are not important because infinite accuracy is never required. Finding the "important" failure states is a principal task in computing system reliability, and several algorithms [4–6, 9–11] have been developed to accomplish that task. These algorithms may require days of computing time for even medium-sized systems, however, and our principal objective in developing a new algorithm was to develop a faster method with which very large systems could be addressed. The SHORTCUT algorithm finds the important cut sets of any noncoherent fault tree, and its principal operations are outlined in the flow chart of Figure 5.

In this chapter we will discuss each of the blocks of ure 5. We confine the discussion of blocks 1 through 5 to this introductory section because they were addressed in earlier sections. Individual sections will be devoted to the cut set reduction problem (Section 3.2), to cut set truncation (Section 3.3), and to the bottom-up substitution process (Section 3.4).

3.1.1 TREE MODULARIZATION AND MODULE SELECTION

Referring to Figure 5, recall that a *module* in a fault tree (the *parent*) is a fault tree whose gates and their interconnections are taken from the parent tree, and which can be treated as independent from the rest of the parent tree. The top event of a module may thus be viewed as a basic event of the parent tree. The existence of modules considerably simplifies the evaluation of cut sets and the calculation of the probability of the top event of the parent tree.

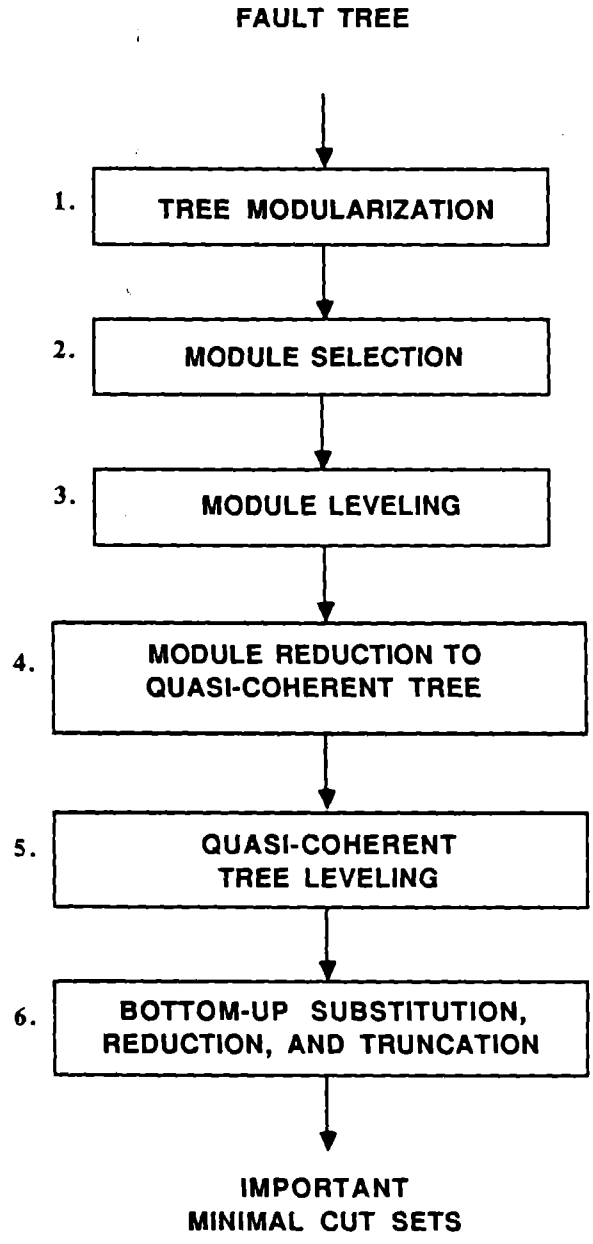


Figure 5. Simplified view of the SHORTCUT algorithm.

Many tree modularization methods exist [12], and these will not be discussed here. The choice of which module is processed first is usually arbitrary, or at least very problem dependent, and it will also not be treated here.

3.1.2 MODULE LEVELING

Fault trees or their modules need not be leveled to apply the methods presented in this report, but a leveled tree is easier to visualize and to process, and so it is usually worth some effort searching for a tree's levels, if it has any. The process of leveling the module was discussed earlier, and a method was briefly described in Section 2.2.6.

3.1.3 MODULE REDUCTION AND LEVELING

Reducing the chosen module to a quasi-coherent tree is accomplished by the COHERE algorithm, which was presented in the previous section. In Block 5 of Figure 5, the resulting tree is itself leveled by a method such as that discussed in Section 2.2.6.

3.2 Reducing Cut Sets: The SUBSET Algorithm

During the cut set extraction process, many sets are generated which are subsets of other sets, or which contain other sets. Since $P(A \cup B) = P(B)$ whenever any set B contains another set A, there is no point storing A for further processing since it contributes nothing to the final answer sought. Only *minimal sets* need to be retained, and nonminimal sets must be *reduced* to minimal ones.

It turns out that most of the computational work required to extract cut sets is expended in this reduction process, because $\mathcal{O}(m^k n^\ell)$ work must be expressed to reduce m sets which are n-long, where k and ℓ are exponents whose size depends on the efficiency of the reduction procedure. Indeed, there are typically $\mathcal{O}(2^n)$ cut sets that must be reduced, and thus $\mathcal{O}(2^{mk+\ell})$ work may be required: an enormous quantity which is very sensitive to the value of k. Since k is essentially determined by the reduction procedure, it is *crucial* that this procedure be efficient.

Typical procedures [5,6,11] require $\mathcal{O}(m^2 n)$ work, so that $k = 2$ and $\ell = 1$ for these methods. The purpose of this section is to present the SUBSET algorithm, which is only *linear* in m, *i.e.*, for which $k = 1$. When $m = 100$, for instance, SUBSET is thus 100 times faster than other methods in reducing the m sets, and 2^n times faster in computing all the cut sets, whatever the value of n.

Using bit matrix and bit vector operations, SUBSET is considerably faster than any other method, although this speed-up is somewhat dependent upon the word length of serial computers used.

3.2.1 THE *SUBSET* ALGORITHM

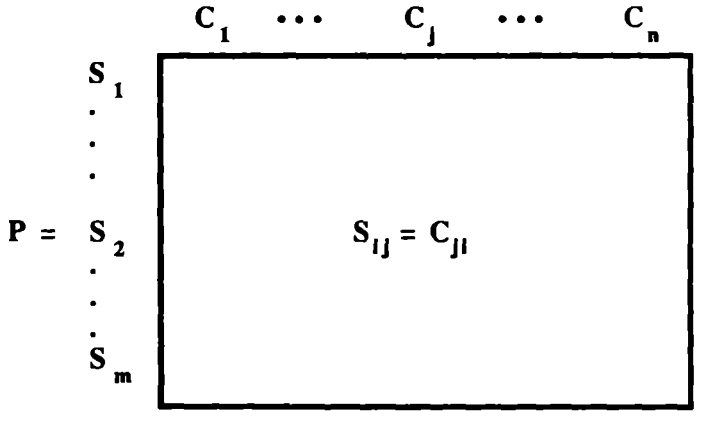
The principles underlying SUBSET can be informally described as follows.

Starting with a collection (problem set) P of cut sets, and for any cut set $S_i \in P$, P is partitioned into an increasing class $\{S_j^i\}_{\not\subset}$ of sets in P which cannot be contained in S_i at step j, and into a decreasing class $\{S_j^i\}_{\subset}$ which can be contained in S_i at step j, $j = 1, 2, \ldots, k_i$. If, for any step j, $\{S_j^i\}_{\not\subset} = P - S_i$ $\left(\text{or } \{S_j^i\}_{\subset} = \emptyset\right)$, we are done, and S_i contains no sets in P (except itself, of course). If there is no such j, then $\{S_{k_i}^i\}_{\subset}$ is the class of sets contained in (absorbed by) S_i, and these may be discarded.

We start with the coherent case, and we conclude with its extension to the noncoherent case.

3.2.2 *SUBSET:* THE COHERENT CASE

Consider a set of simple sets P represented as a matrix:



with rows (simple sets) $S_i = (S_{i1}, \ldots, S_{ij}, \ldots, S_{in})$, and columns $C_j = (C_{j1}, \ldots, C_{ji}, \ldots, C_{jm})$. In this matrix, entries in row i and column j are denoted by S_{ij}, $S_{ij} \in \{0, 1, -, \emptyset\}$, and mean the following in the associated fault tree:

$S_{ij} = 0$: component j is failed in cut set i.

$S_{ij} = 1$: component j is working in cut set i.

$S_{ij} = -$: component j is irrelevant to cut set i.

$S_{ij} = \emptyset$: cut set i is empty.

Now define the following:

$Z_i = \langle \{j\colon\ S_{ij} = 0\}, \leq \rangle$, the set of column indices whose associated columns have a zero entry in position (row) i, ordered by the standard inequality ordering $\leq$.

$Z_j = \langle \{i\colon\ C_{ji} = 0\}, \leq \rangle$, the set of row indices whose associated rows have a zero entry in position (column) j, ordered by the standard inequality ordering $\leq$.

Similarly for $W_i = \langle \{j\colon\ S_{ij} = 1\}, \leq \rangle$ and $W_j = \langle \{i\colon\ C_{ji} = 1\}, \leq \rangle$.

For two nonempty sets S_p and S_q in P, the following simple fact can be easily derived from earlier definitions and properties of simple sets.

FACT.

$$S_p \not\subset S_q \text{ iff } (\exists\, k) \ni \left(S_{qk} = 0 \bigwedge S_{pk} \neq 0\right) \bigvee \left(S_{qk} = 1 \bigwedge S_{pk} \neq 1\right) \quad . \tag{22}$$

As discussed in our introduction, Equation (22) is the basis for the SUBSET algorithm whose flowchart we now describe in Figure 6.

3.2.3 AN EXAMPLE

Consider the problem set

$P =$

	C_1	C_2	C_3	C_4	C_5
S_1	0	—	0	—	—
S_2	—	—	0	0	—
S_3	0	0	—	—	—
S_4	—	0	—	0	0
S_5	—	0	—	0	—

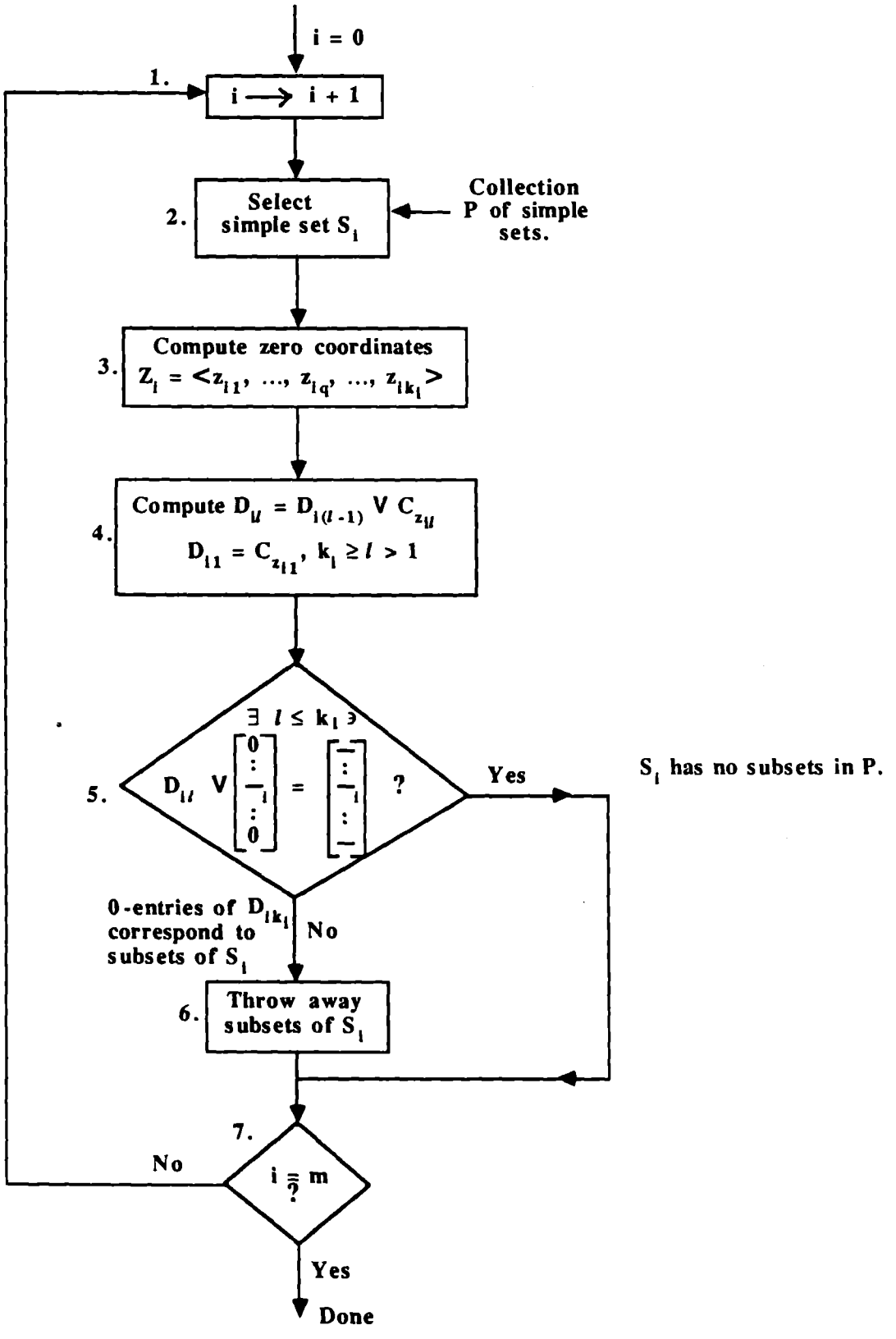


Figure 6. Flow chart of the SUBSET algorithm.

Then, for S_1: $Z_1 = \langle 1, 3 \rangle$

$$D_{12} = \begin{bmatrix} 0 \\ - \\ 0 \\ - \\ - \end{bmatrix} \bigvee \begin{bmatrix} 0 \\ 0 \\ - \\ - \\ - \end{bmatrix} = \begin{bmatrix} 0 \\ - \\ - \\ - \\ - \end{bmatrix} ,$$

and S_1 has no subsets.

For S_2: $Z_2 = \langle 3, 4 \rangle$

$$D_{22} = \begin{bmatrix} 0 \\ 0 \\ - \\ - \\ - \end{bmatrix} \bigvee \begin{bmatrix} - \\ 0 \\ - \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} - \\ 0 \\ - \\ - \\ - \end{bmatrix} ,$$

and S_2 has no subsets.

For S_3: $Z_3 = \langle 1, 2 \rangle$

$$D_{32} = \begin{bmatrix} 0 \\ - \\ 0 \\ - \\ - \end{bmatrix} \bigvee \begin{bmatrix} - \\ - \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} - \\ - \\ 0 \\ - \\ - \end{bmatrix} ,$$

and S_3 has no subsets.

For S_4: $Z_4 = \langle 2, 4, 5 \rangle$

$$D_{42} = \begin{bmatrix} - \\ - \\ 0 \\ 0 \\ 0 \end{bmatrix} \bigvee \begin{bmatrix} - \\ 0 \\ - \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} - \\ - \\ - \\ 0 \\ 0 \end{bmatrix} ,$$

and S_5 is possibly a subset of S_4 at Step 2.

$$D_{43} = D_{42} \bigvee C_5 = \begin{bmatrix} - \\ - \\ - \\ 0 \\ 0 \end{bmatrix} \bigvee \begin{bmatrix} - \\ - \\ - \\ 0 \\ - \end{bmatrix} = \begin{bmatrix} - \\ - \\ - \\ 0 \\ - \end{bmatrix} \quad ,$$

and S_4 has no subsets.

For S_5: $Z_5 = \langle 2, 4 \rangle$

$$D_{52} = \begin{bmatrix} - \\ - \\ 0 \\ 0 \\ 0 \end{bmatrix} \bigvee \begin{bmatrix} - \\ 0 \\ - \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} - \\ - \\ - \\ 0 \\ 0 \end{bmatrix} \quad ,$$

and S_4 is a subset of S_5, and may be thrown away.

3.2.4 THE NONCOHERENT CASE

Extension to noncoherent problems is simple, as shown in the following extension of the flowchart of Figure 7.

3.2.5 THE COMPLEXITY OF *SUBSET*

Using bit vector implementations, such as those available on Cray computers, the complexity of SUBSET is linear in the number m of simple sets, and in the number n of basic events or components of the system. Thus, $C(\text{SUBSET}) = \mathcal{O}(n+m)$, a considerable improvement over other methods such as FTAP [10] and SETS [5], as shown in Figure 8. Such methods typically require $\mathcal{O}(m^2 n)$ computations, a significant increase. If we consider a small problem where $m = 1000$, $n = 100$, SUBSET requires $\mathcal{O}(10^5)$ computations, whereas other methods require $\mathcal{O}(10^8)$ computations, a much larger number. Actually, the few tests which produced Figure 8 yielded a *sublinear* behavior, *i.e.*, the complexity was $\mathcal{O}(\alpha n + \beta m)$, $\alpha, \beta < 1$. Our previous example, for instance, required only 16 bit vector operations, whereas other methods would require $\mathcal{O}(m^2 n) = \mathcal{O}(125)$ operations.

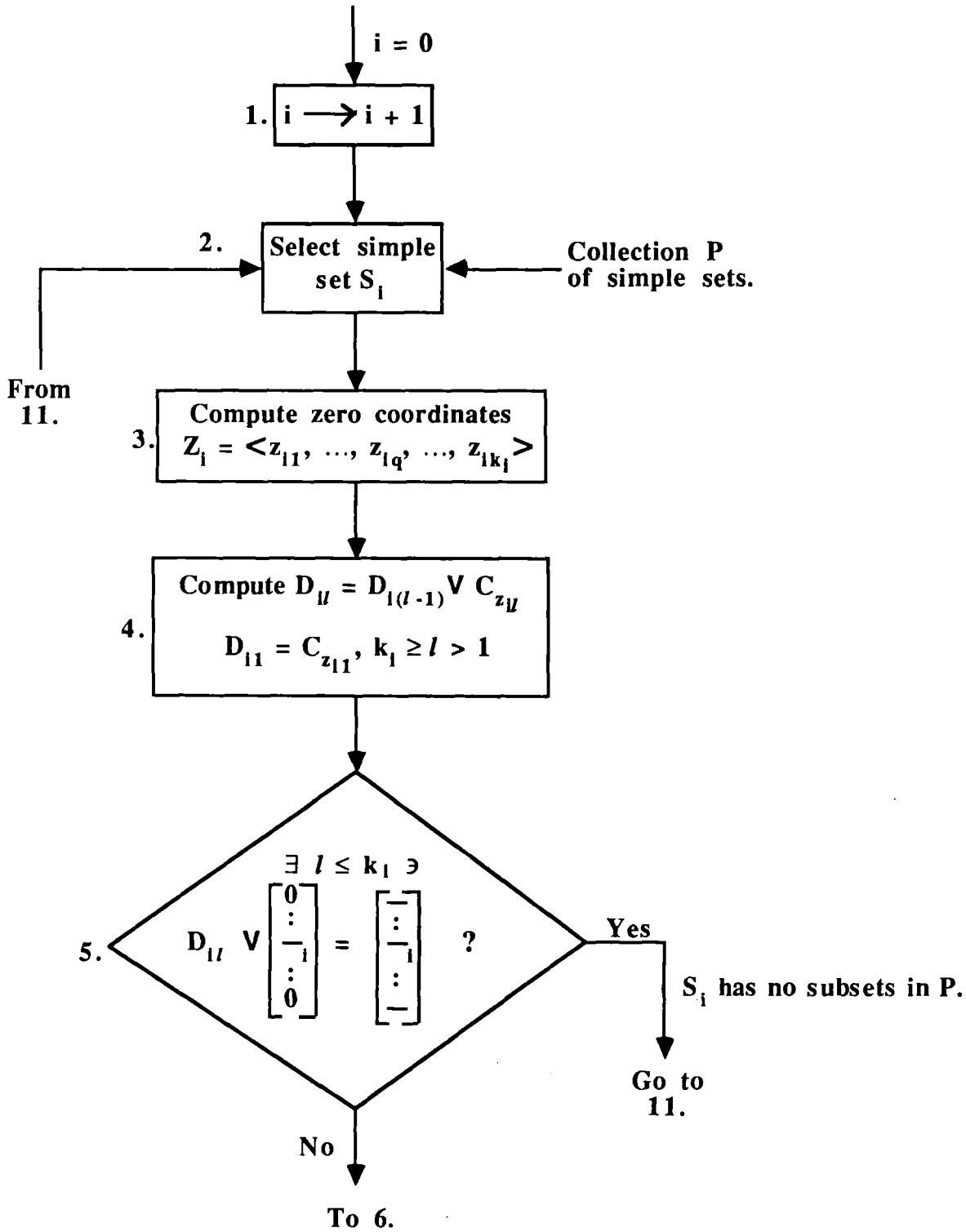


Figure 7. (continued)

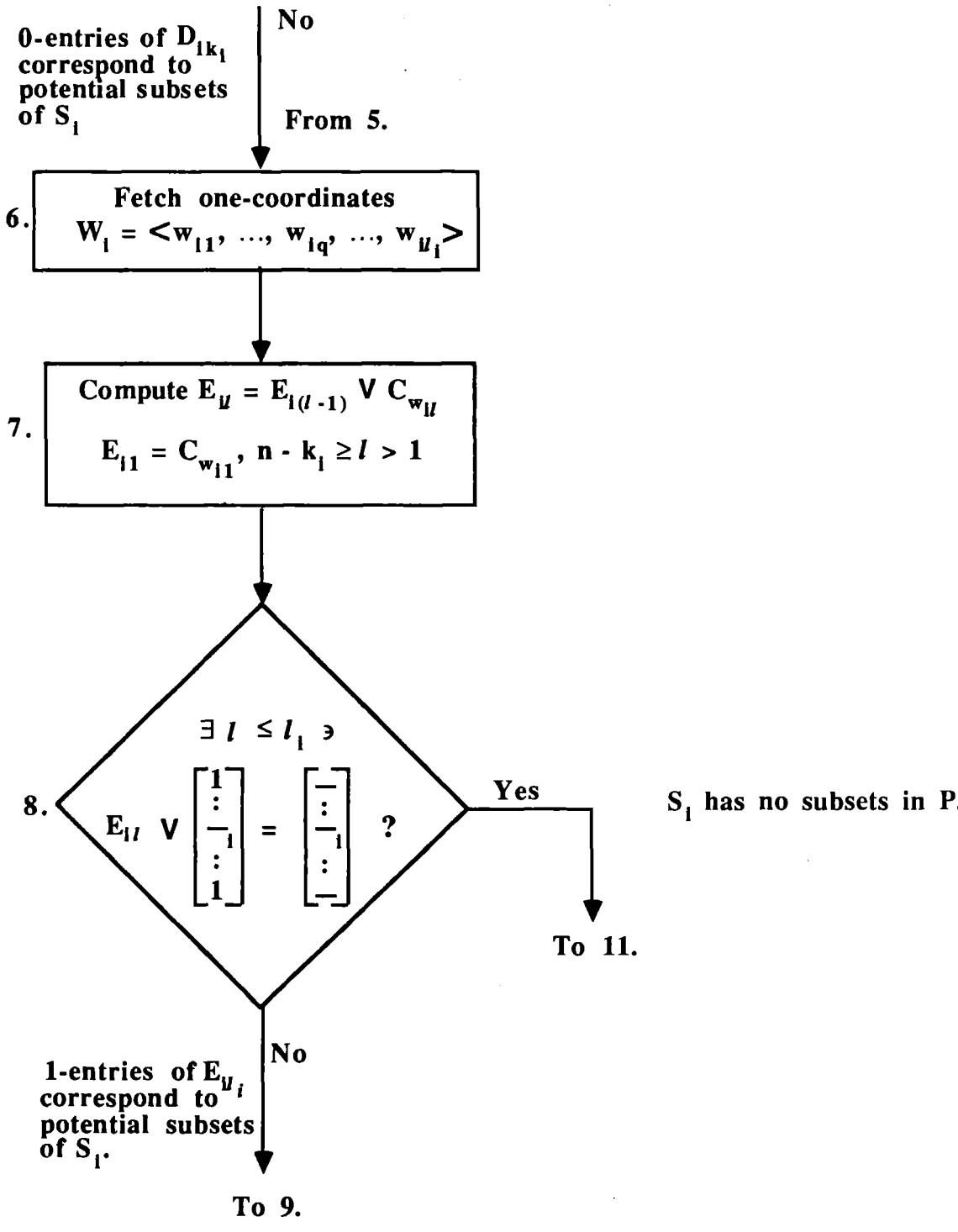


Figure 7. (continued)

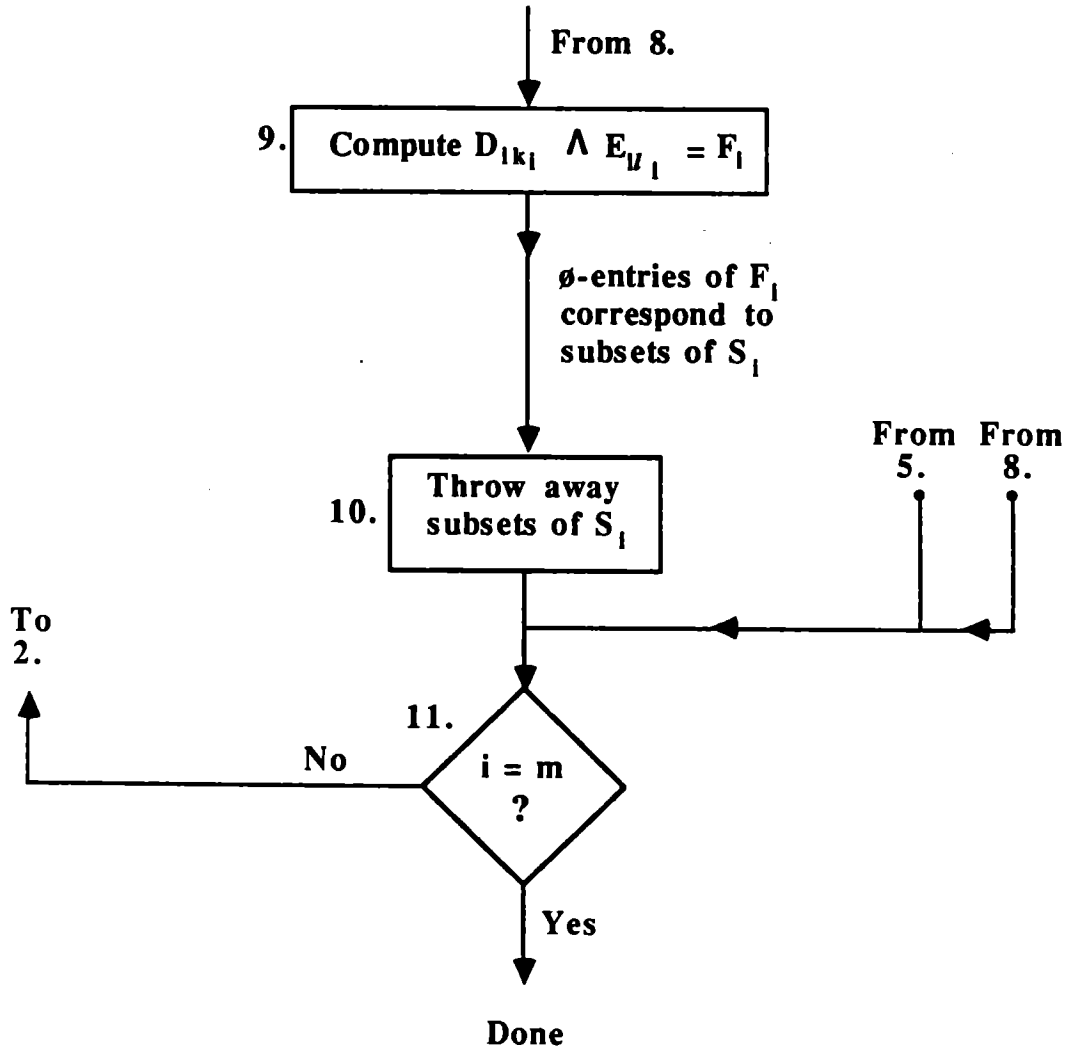


Figure 7. Flow chart of the SUBSET algorithm for the noncoherent case.

This will, in fact, be the case in most practical systems, because $\#(Z_i)$, the number of elements in the zero sets Z_i, will usually be much less than the number n of components in the system. Furthermore, as subsets are found, they are removed from P, and thus, progressively fewer simple sets in P have to be examined. Observe that $\beta = 1$ only if there are no subsets. Also observe that $\alpha = 1$ only if all sets in P consist only of zeros, an obviously impossible situation.

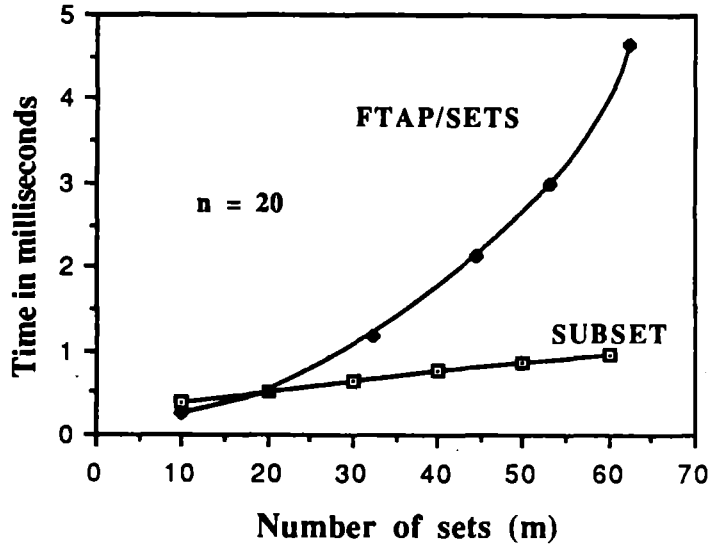


Figure 8. Comparison of SUBSET to other methods.

3.3 Cut Set Truncation: The TRUNC Algorithm

One effective way to speed up the computation of cut sets is to retain only those expressions which are less then k-long, *i.e.*, have fewer than k literals. Then, unless the dual tree is used as discussed in Chapter 1, only at most $2^{k/2}$ expressions may need to be found, but then only a lower bound on the system failure probability can be obtained since it is impossible to assess the quantitative effects of throwing away expressions. Things are even worse for noncoherent trees, since throwing away expressions or disjuncts as the top event is approached may in fact *increase* the final probability. Throwing away a complemented expression is equivalent to *adding* another expression, thereby increasing the probability. This was the main argument for developing the COHERE algorithm, which reduces the noncoherent tree *before* any approximations or truncations are made. It was also the principal reason for computing the dual tree since then we get both an upper *and* a lower limit on the correct failure probability.

Whereas various criteria for truncating expressions are discussed below, the current version of SHORTCUT only includes a simplified version TRUNC, which throws away expressions on the basis of length only. Such truncation is done at every AND gate and limits every expression to be at most k-long, thereby assuring that *all* minimal cut sets

which are at most k-long are retained by the time the top event is reached. Observe that, since a simple set does not qualify as a cut set candidate until the top of the tree has been reached, expressions which are discarded by this process are not cut sets, but only ordinary simple sets.

Current methods reject simple sets either on probabilistic or on set-theoretic grounds. Typically, a set is probabilistically rejected if its probability is less than a given rejection threshold. Usually, basic events are assumed independent in this calculation and a set's probability is simply assumed to equal the product of the probabilities of its events. A more tractable and popular method is to reject a set if its set-theoretic "size" (in the sample space) is smaller than a reference size. A good measure of the smallness of a set is the "length" of the set, that is the number of its non-don't care entries. The *length* of a Boolean expression with no redundant symbols, terms, or literals is thus simply the number of literals appearing in the expression, and any expression exceeding a given length is thus set-theoretically too small to be retained.

A careful analysis of both methods reveals that they can be very misleading, and can lead to retaining too many sets and to rejecting important ones. The principal reason is that it is not the individual probability or size of sets that matters, but the contribution they make to the overall system reliability estimate. This is a differential sensitivity issue which can only be settled when the effects of rejecting *any* combination of sets are known, an obviously impossible task.

We attempt to approximate such an approach in this section, only to determine where it leaves us in revealing weaknesses of current methods in a quest for better ways. It was with some relief that we discovered that current truncation methods are in fact consistent with the correct view, at least in the first order, and we demonstrate that in the next section.

3.3.1 THE TRUNCATION RULE *TRUNC*

Our method is based on the fact that the contribution of a set B to a union $A \cup B$ relative to a probability measure P is simply $P(B - A)$, the probability of the set of points in B not in A.

When a collection $\{S_i\colon\ i = 1, \ldots, m\}$ is considered, we rank each set S_i in accordance

with its contribution to the total union $\cup_{i=1}^{m} S_i$, which we approximate by the "average" contribution $F(S_i) = m^{-1} \sum_{j=1}^{m} P(S_i - S_j)$.

The probabilities $P(S_i - S_j)$ are difficult to compute, however, because the contributions of S_i over S_j in each coordinate are not disjoint. But recall from our mathematical introduction that

$$S_i - S_j = \bigcup \left\{ S_i \bigcap \overline{E}_{jk} \colon\ k \in \mathrm{DOM}(S_i/S_j) \right\} \quad , \tag{23}$$

where $\overline{E}_{jk}$ is the complement of the projection of S_j on the kth coordinate. Also recall that $S_j = \cap_{k=1}^{n} E_{jk}$.

We still face the problem that the sets $(S_i \cap \overline{E}_{jk})$ in Equation (23) are not disjoint, and thus $P(S_i - S_j) \neq \Sigma P(S_i \cap \overline{E}_{jk})$. However, a slight transformation in the members of $S_i - S_j$ is sufficient to obtain a disjoint representation of $S_i - S_j$ as we now show. Each set $\overline{E}_{jk}$ may be replaced by the set

$$H_{jk} = \overline{E}_{jk} \bigcap_{\ell<k} \{E_{j\ell} \quad , \quad k, \ell \in \mathrm{DOM}(S_i/S_j)\}$$

without affecting the union $\cup \{S_i \cap \overline{E}_{jk}\}$ defined earlier in Equation (23). The sets H_{jk} are *disjoint*, as we illustrate in the next example.

Example. Consider two sets $A = (1, -, -, 1, -, -, 1)$ and $B = (-, 1, -, -, 1, -, -,)$. Then $\mathrm{DOM}(B|A| = \{1, 4, 7\}$, and

$$B - A = \bigcup \left\{ \begin{array}{l} (0, 1, -, -, 1, -, -) = B_1 \\ (-, 1, -, 0, 1, -, -) = B_2 \\ (-, 1, -, -, 1, -, 0) = B_3 \end{array} \right\}$$

If we substitute two sets $B_2' = (1, 1, -, 0, 1, -, -)$ and $B_3' = (1, 1, -, 1, 1, -, 0)$ for B_2 and B_3, respectively, we get

$$B - A = \bigcup_{i=1}^{3} B_i = \sum_{i=1}^{3} B_i' \quad , \tag{24}$$

where $B_1' = B_1$.

The sets B_i' are thus disjoint and we use the symbol Σ to denote disjoint unions. Returning to the H_{jk}'s, we now get (indices not important).

$$\bigcup H_{jk} = \sum H_{jk} \quad , \tag{25}$$

and

$$\begin{aligned} S_i - S_j &= \bigcup \left\{ S_i \bigcap H_{jk} \colon\ k, \ell \in \mathrm{DOM}(S_i/S_j) \right\} \\ &= \sum \left\{ S_i \bigcap H_{jk} \colon\ k, \ell \in \mathrm{DOM}(S_i/S_j) \right\} \\ &= \sum \left\{ S_i \bigcap \overline{E}_{jk} \bigcap_{\ell<k} \{E_{j\ell} \colon\ k, \ell \in \mathrm{DOM}(S_i/S_j)\} \right\} \quad . \end{aligned} \tag{26}$$

We have obtained a convenient form for the probability of $S_i - S_j$:

$$P(S_i - S_j) = \sum \left\{ P\left(S_i \bigcap \overline{E}_{jk} \bigcap_{\ell<k} \{E_{j\ell} \colon\quad k, \ell \in \mathrm{DOM}(S_i/S_j)\} \right) \right\} \quad . \tag{27}$$

If the basic events (variables)—and thus the elementary simple sets—are independent,

$$P\left(S_i \bigcap \overline{E}_{jk} \bigcap_{\ell<k} E_{j\ell} \right) = P(S_i) P(\overline{E}_{jk}) \prod_{\ell<k} \{P(E_{j\ell}) \colon\ k, \ell \in \mathrm{DOM}(S_i/S_j)\} \, . \tag{28}$$

Combining Equations (28) and (27),

$$P(S_i - S_j) = \sum \left\{ P(S_i) P(\overline{E}_{jk}) \prod_{\ell<k} \{P(E_{j\ell}) \colon\ k, \ell \in \mathrm{DOM}(S_i/S_j)\} \right\} \quad . \tag{29}$$

But $P(\overline{E}_{jk}) \approx 1$, at least for reliable coherent systems, and $\prod_{\ell=1}^{L} P(E_{j\ell}) \ll P(E_{j1})$ for typical systems, reliable or not. Thus

$$P(S_i - S_j) \approx P(S_i) \quad , \tag{30}$$

and

$$F(S_i) = \frac{1}{m} \sum_{j=1}^{m} P(S_i - S_j) \doteq P(S_i) \quad . \tag{31}$$

This formally confirms that, for coherent and reliable systems, the probabilities of a simple set are an excellent measure of the contribution of the set to the overall probability.

For noncoherent systems, $P(\overline{E}_{jk}) \not\approx 1$ and $\prod_{\ell=1}^{L} P(E_{j\ell}) \not\ll P(E_{j1})$ necessarily, and all the terms in Equation (30) reduce to the form

$$F(S_i) = \frac{1}{m} \sum_{j=1}^{m} P(S_i - S_j) \quad . \tag{32}$$

When basic variables or events E_i are independent, and when their probabilities are comparable in magnitude, $P(E_\ell \cap E_k) \ll P(E_\ell)$ usually, particularly if the associated system is at least moderately reliable. In such cases, the shortness of a simple set is a good indicator of that set's relative importance, where a simple set is *shorter* than another if it has more don't cares, or *less* non-don't cares. Recall that a shorter simple set is usually *larger* set-theoretically than a longer simple set, *i.e.*, has a larger probability value. Thus, an effective and accurate way to rank simple sets—and to reject simple sets during the cut-set computation process—is to rank them by length, shortest sets first. Since the SHORTCUT algorithm can only increase the length of simple sets as the top variable (top of the tree) is approached, limiting sets to those of length at most k will guarantee that all cut sets of length at most k are produced, and no more, and this is precisely what algorithm TRUNC does.

3.4 Bottom-Up Substitution: The SHORTCUT Algorithm

The SHORTCUT algorithm is a bottom-up substitution algorithm [6,11] which replaces intermediate events in a leveled quasi-coherent fault tree with an expression involving basic events only, in accordance with the Boolean operations encountered at each gate of the tree. Starting with level 1, each gate is replaced by the appropriate expression for its output until the top variable or event is reached. During this process, various reduction and truncation steps are taken at each gate to avoid carrying along unnecessary simple sets. These steps are based on five Boolean reduction rules or tautologies:

R_1 : $A\cdot\overline{B} + A\cdot B = A$ (Exhaustion)

R_2 : $A\cdot B + A = A$ (Absorption)

R_3 : $A\cdot A = A$ (Idempotence)

R_4 : $\overline{\overline{A}} = A$ (Double Negation)

R_5 : $A\cdot B\cdot\overline{A} = \emptyset$ (Exclusion)

Even in moderately complicated noncoherent fault trees, intermediate expressions rarely have the form suitable for applying rule R_1, and this rule was not built into our procedure.

The entire procedure may be described by what is done at each OR gate and at each AND gate. In Section 3.4.1, we discuss OR gate operations, and in Section 3.4.2., we discuss AND gates. Finally in Section 3.4.3., we present a flowchart of SHORTCUT.

3.4.1 OPERATIONS AT *OR* GATES

Consider some OR level $G_\ell \subset G^+$ in a quasi-coherent leveled tree. The jth gate $g_{\ell j}$ at that level has $k(\ell, j)$ inputs $I_{\ell j} = \{I_{\ell jk}\colon\ k = 1, \ldots, k(\ell, j)\}$.

Each input is itself a reduced collection $S_{\ell jk} = \{S_{\ell jkq}\colon\ q = 1, \ldots, q(\ell, j, k)\}$ of simple sets, and these collections must now be *collectively* reduced using the rule $A\cdot B + A = A$ embodied in SUBSET. The only operation at each OR gate is to reduce the collection of simple sets involved in the union

$$O_{\ell j} = \bigcup_{k=1}^{k(\ell,j)} \bigcup_{q=1}^{q(\ell,j,k)} S_{\ell jkq}$$

which represent the output of the gate. Once reduced, this collection replaces the OR gate and the process continues with the next OR gate $g_{\ell(j+1)}$ until level G_ℓ is completed.

Observe that, in a typical situation, there is no need to compare or reduce sets associated with any one terminal, since these have been previously reduced in the tree, as we shall see when we discuss the overall SHORTCUT algorithm. Thus sets at any terminal

must only be reduced relative to sets at another terminal, and this speeds up the reduction process considerably.

No truncation is necessary at OR gates, since the OR operation does not alter the length or size of its input sets. (Rare cases of the form $A \cdot B + A \cdot \overline{B}$ are neglected here.)

3.4.2 OPERATIONS AT *AND* GATES

Consider now some AND level $G_\ell \subset G^*$ and its jth gate $g_{\ell j}$ with its $k(\ell, j)$ inputs $I_{\ell j} = \{I_{\ell jk}\colon\ k = 1, \ldots, k(\ell, j)\}$. Similarly to OR gates, each input is itself a reduced collection $S_{\ell jk} = \{S_{\ell jkq}\colon\ q = 1, \ldots, q(\ell, j, k)\}$, as shown in Figure 9.

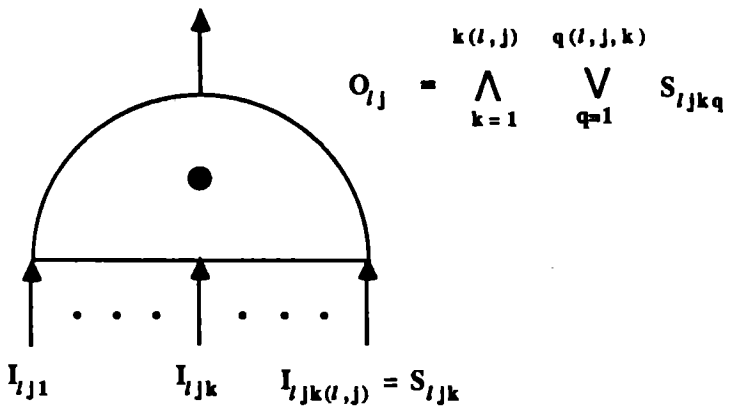


Figure 9. AND gates transform reduced collections of simple sets into their product.

Input collections ("terminals") are AND-ed pairwise, smallest collections first. The result of AND-ing two collections is a new collection whose length is considered in selecting a new pair, although the product of two collections will usually be larger than the collections yet to be multiplied. Consider two such collections $S = \{S_1, \ldots, S_u\}$ and $T = \{T_1, \ldots, T_v\}$. Then $\cup_{j=1}^{v} T_j \cap \cup_{i=1}^{u} S_i = \cup_{j=1}^{v} \cup_{i=1}^{u} T_j \cap S_i$, and this reveals our approach. Using bit vector operations, u new collections $\{T_j S_1, \ldots, T_j S_u\}$ are produced,

each obtained by operating on the columns of S, one column at a time, as we did in the SUBSET algorithm. Let us illustrate that with an example.

EXAMPLE. Let

$S =$

	c_1	c_2	c_3	c_4	c_5
S_1	—	0	—	1	—
S_2	1	0	0	—	—
S_3	—	—	1	—	—
S_4	1	—	1	—	1

and

$T =$

	d_1	d_2	d_3	d_4	d_5
T_1	1	—	—	0	1
T_2	1	1	—	1	0
T_3	—	1	—	1	1
T_4	1	—	1	—	—

Then $S \cdot T = (S_1 \cup S_2 \cup S_3 \cup S_4) \cdot (T_1 \cup T_2 \cup T_3 \cup T_4) = T_1S \cup T_2S \cup T_3S \cup T_4S$, and four collections of sets are produced, one for each T_j. But before we show how these products are computed, observe that certain sets in S and T are disjoint, like S_1 and T_1, and S_4 and T_2. Products $S_1 \cap T_1$ and $S_4 \cap T_2$ are thus empty and can be thrown away, and this is done first. Using bit vector operations, vectors (columns) c_i and d_j are compared to check for complementing entries. Any set in S which complements at least one set in T in at least one entry is neglected in the corresponding product $T_j \cap S$. Consider only the collection $T_1 \cap S = T_1 \cap \{S_1, \ldots, S_4\}$. Then $T_1 \cap S = T_1 \cap (S - S_1)$, a smaller collection. Recalling the definition of products of simple sets, $T_1 \cap S$ is thus produced as follows:

1. Remove S_1 from S, obtaining $S' = \{S_2, S_3, S_4\}$.

2. Set coordinate c_1 of S' to 1, the value of the first coordinate of $T_1(d_1)$.

3. Set coordinate c_4 of S' to 0, the value of the fourth coordinate (d_4) of T_1.

4. Set coordinate c_5 of S' to 1, the value of the fifth and last coordinate of T_1.

5. Truncate the new collection $T_1 \cap S'$ with TRUNC.

6. Reduce resulting collection with SUBSET.

7. Store these minimal sets as collection S_{T_1}.

8. Continue with T_2.

Note also that individual products $T_j \cap S_i$ never need to be computed by this method, resulting in a considerable complexity reduction. Indeed, while other methods require $\mathcal{O}(m_S m_T n)$ operations for multiplying (AND-ing) S and T, our method requires only $\mathcal{O}(m_T n)$ operations, a reduction of orders of magnitude (n is the number of basic variables (events) in the system).

Note also that there is no need for an additional idempotency test $A \cdot A = A$, since this is done automatically in our simple set framework.

Returning to the general case, our example illustrates what is done and we conclude our discussion in the next section where a flowchart of SHORTCUT is presented.

3.4.3 FLOW CHART OF *SHORTCUT*

In this section, we present a flowchart of the overall SHORTCUT algorithm, which is shown in Figure 10. For simplicity, we use the notation and subroutines developed earlier, and we provide below any comments appropriate for the blocks in the flowchart.

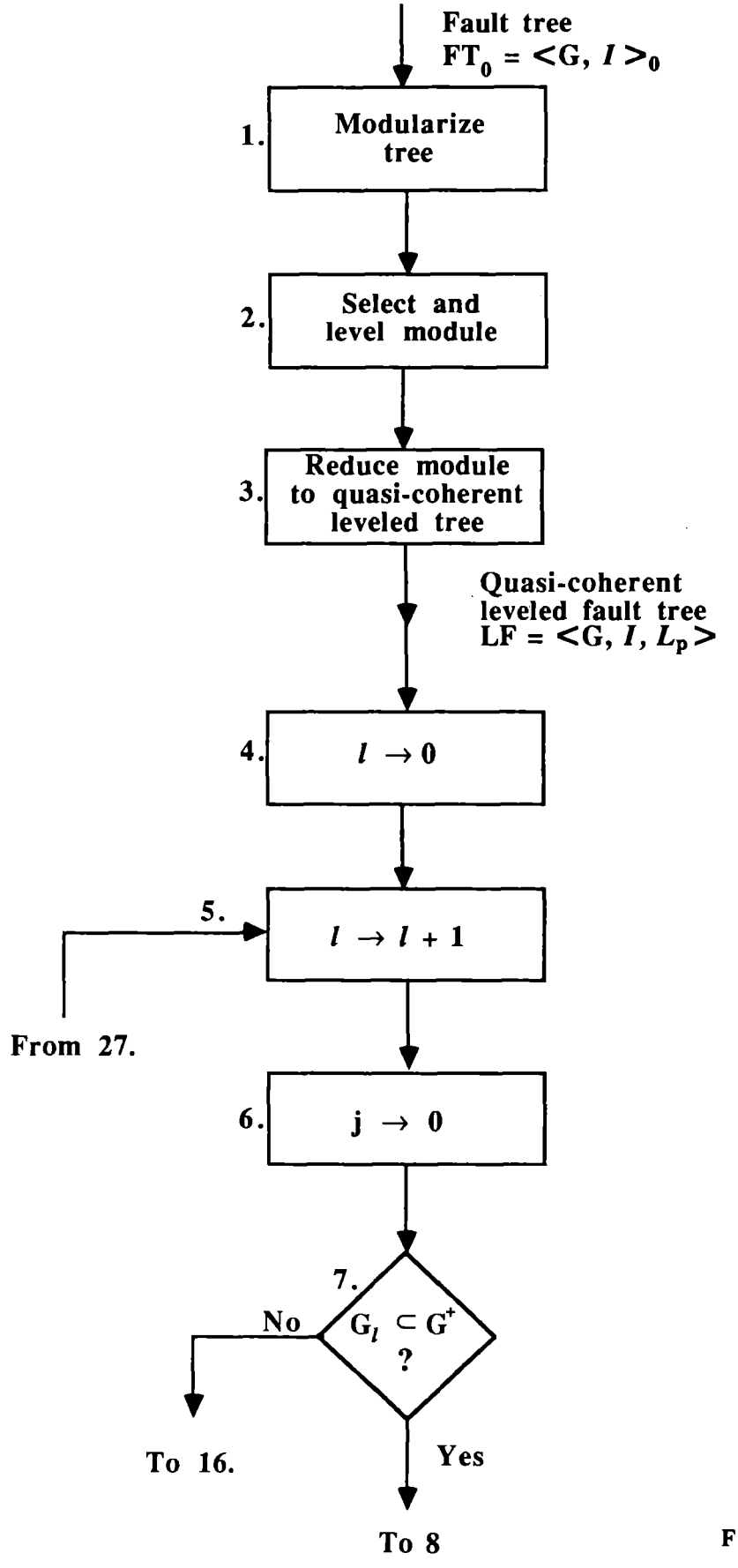


Figure 10. (continued)

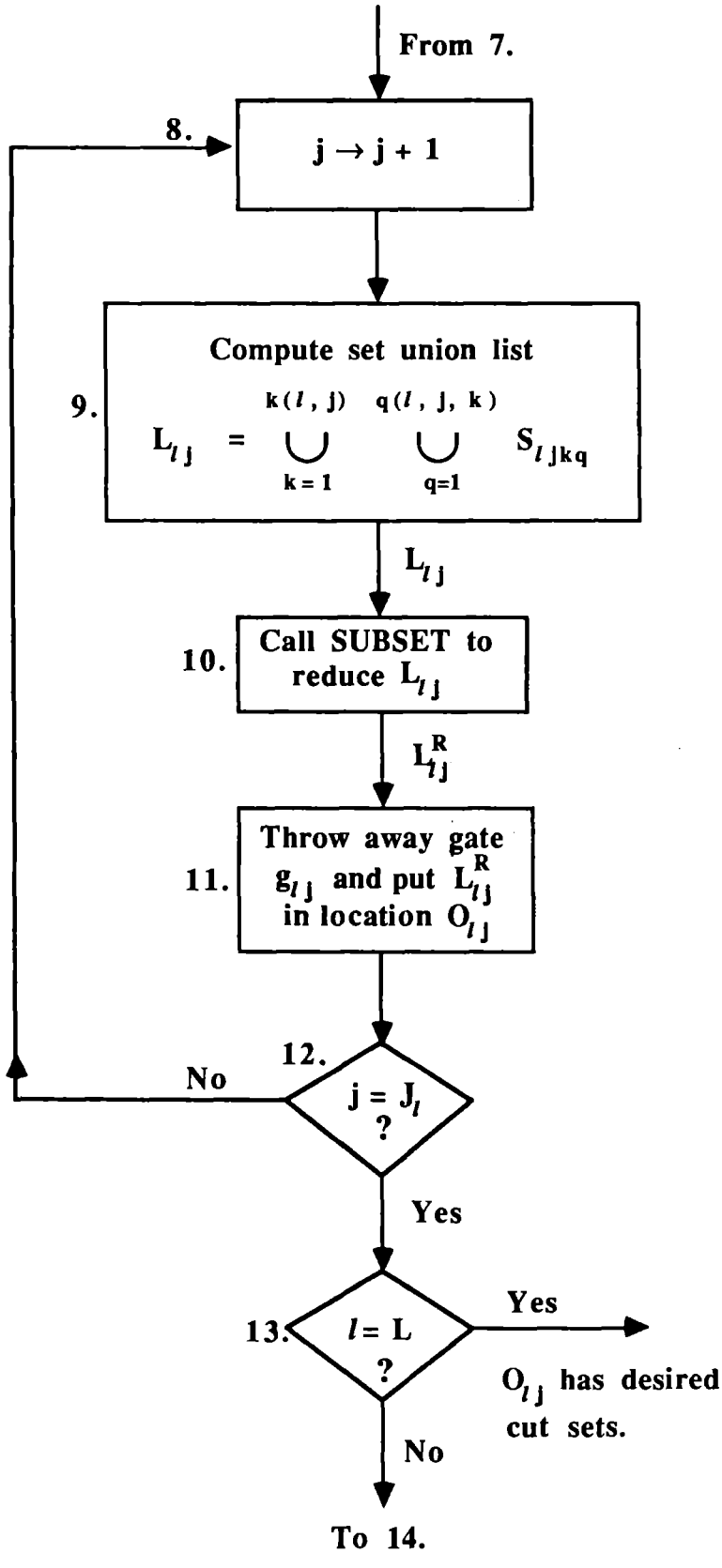


Figure 10. (continued)

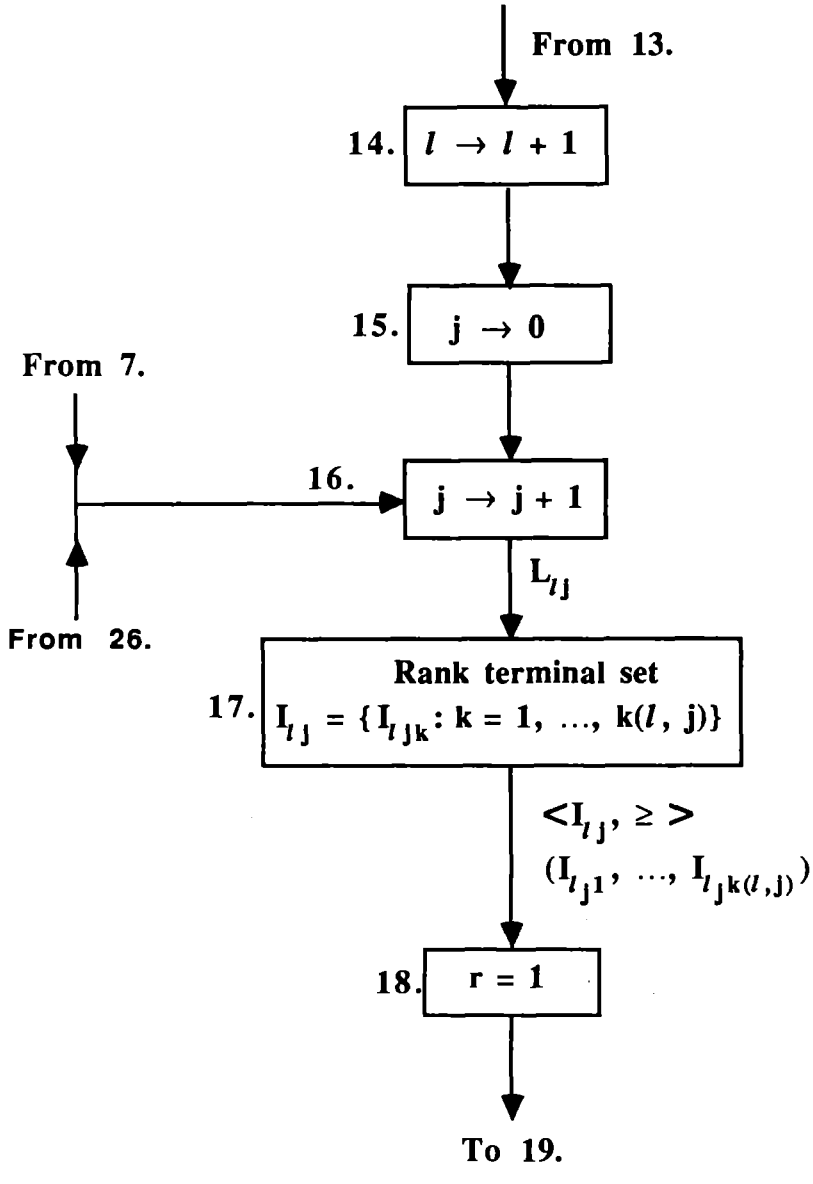


Figure 10. (continued)

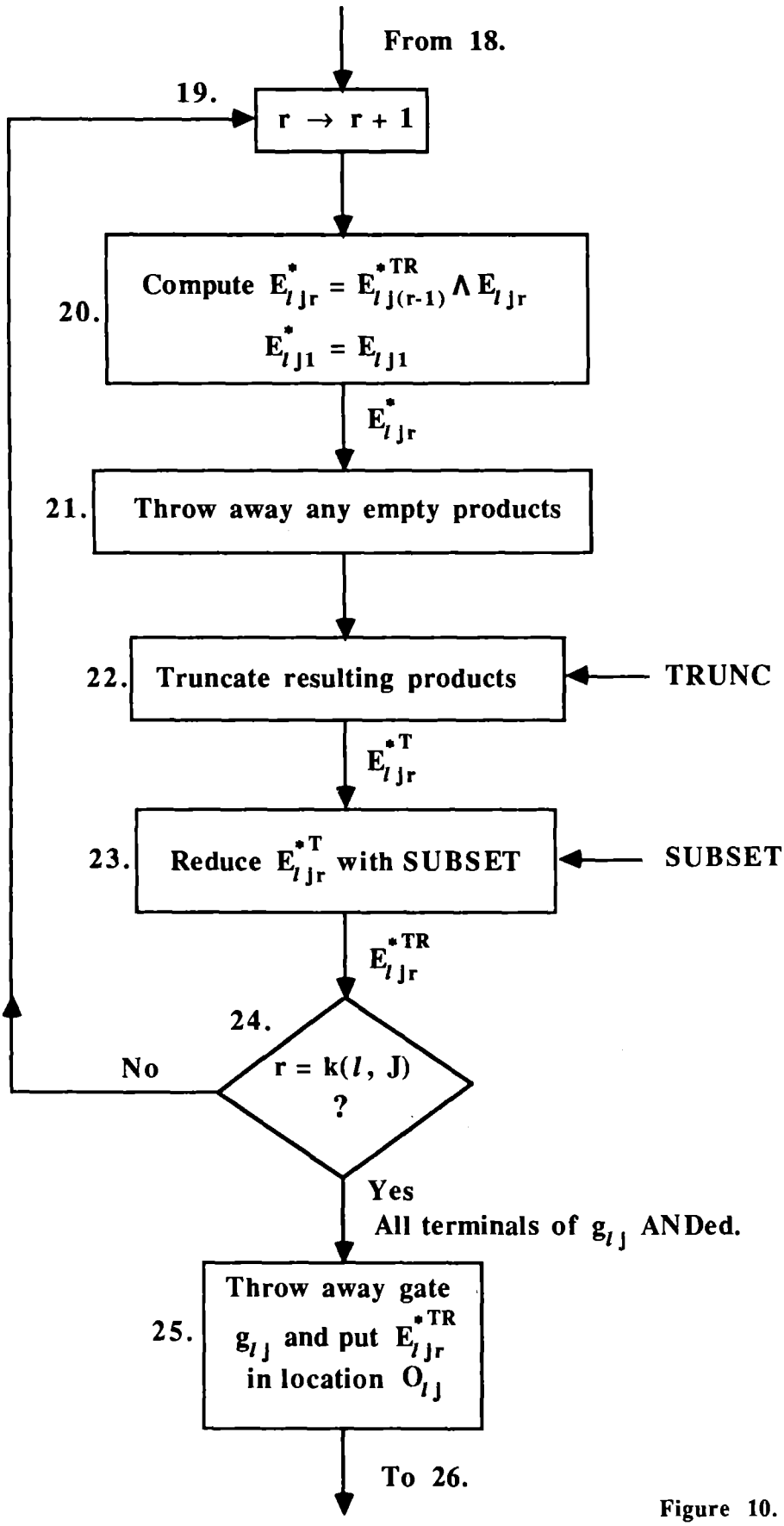


Figure 10. (continued)

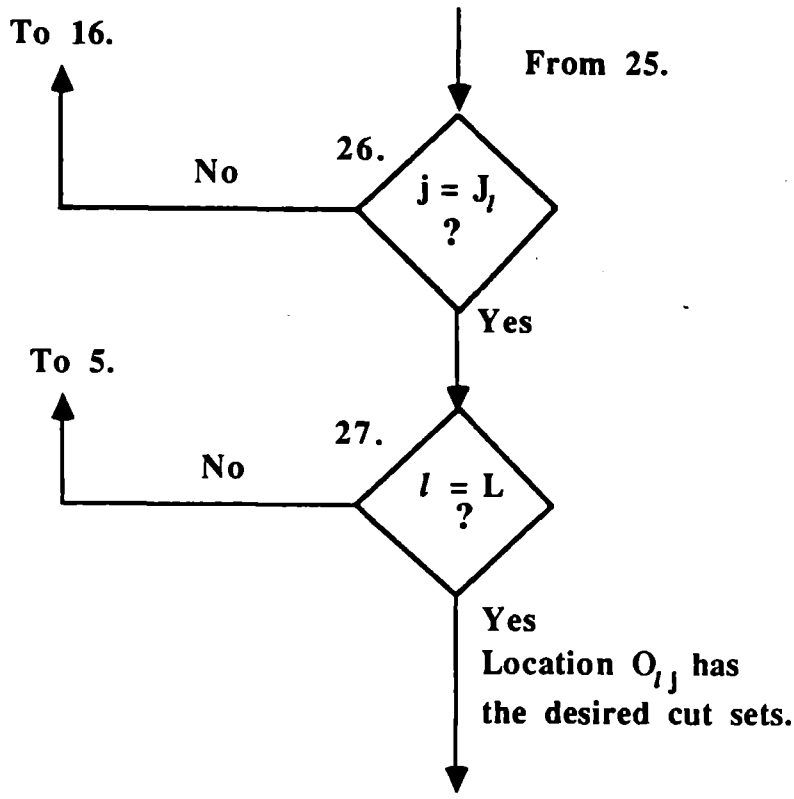


Figure 10. Flow chart of the overall SHORTCUT algorithm.

Block 2

$G = G^+ \cup G^*$, $\mathcal{L}_F = \{G_\ell : \ \ell = 1, \ldots, L\}$ and each level G_ℓ has gates $G_\ell = \{g_{\ell 1}, \ldots, g_{\ell j}, \ldots, g_{\ell J_\ell}\}$.

Block 7

Is level under consideration an OR level?

Block 8

Each gate $g_{\ell j}$ has set of inputs (terminals) $I_{\ell j} = \{I_{\ell jk}: \ k = 1, \ldots, k(\ell, j)\}$.

Block 9

$S_{\ell jkq}$ is the qth simple set (disjunct) at the kth terminal (input) of gate j at level ℓ.

Block 10

Observe that we have not incorporated an inner loop to prevent sets at a given terminal of the OR gate from being compared (again) against each other, as we suggested in Section 3.4.1. Clearly, if a gate has many terminals with only a few sets at each terminal, such a test and control loop would considerably lessen the reduction work required at the gate. This feature will be included when SHORTCUT is implemented on the computer.

Block 12

Has last gate at level ℓ been processed?

Block 17

The current ranking criterion is the length $|E_{\ell jr}|$ of expressions $E_{\ell jr}$ at each terminal (input) r. $I_{\ell jr}$ precedes $I_{\ell js}$ iff $|E_{\ell jr}| \leq |E_{\ell js}|$, where $|E_{\ell jx}| \triangleq$ quantity of simple sets (disjuncts) at terminal $I_{\ell jx}$ of gate $g_{\ell j}$.

Block 20

AND expression $E_{\ell jr}$ at terminal r with previously AND-ed, truncated (T) and reduced (R) expression $E_{\ell j(r-1)}$.

Block 23

Same comment as Block 10.

Block 24

Have all AND gate terminals been AND-ed?

Block 26

Have all gates at level ℓ been processed?

Block 27

Has top level of tree been reached?

In the next and final section, we discuss the computational complexity of the SHORTCUT algorithm.

3.4.4 COMPLEXITY OF *SHORTCUT*

In this section, we develop a worst-case expression for the computational complexity of the SHORTCUT algorithm. For this analysis, we make the following fault tree assumptions.

We assume a quasi-coherent fault tree which has N_g^+ OR gates and $N_g^\cdot$ AND gates. The total number of gates in the tree is therefore $N_g = N_g^+ + N_g^\cdot$. Each gate is assumed to have p inputs (p input collections of simple sets, or p "wires"). The length of each simple set (the number of its non-don't care entries) is assumed to equal k, the truncation value specified in advance by the user. Since many sets will be shorter, this is thus a worst-case assumption because, with our method, shorter sets are easier to manipulate than longer ones.

The discussion is presented in three parts. First, we discuss the work required at OR gates (Section 3.4.4.1). Then, we discuss the complexity of processing AND gates (Section 3.4.4.2). Finally, we determine the overall complexity of SHORTCUT in Section 3.4.4.3.

3.4.4.1 Computational Work at OR Gates.

We make the worst-case assumption that all possible k-long sequences (simple sets) are fed to each of the p^+ gate input terminals. There are $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ such sets. For usual fault trees, $n \gg k$, and k is small. Then $\binom{n}{k} \approx \frac{n^k}{k!}$ is a very good approximation, and $\binom{n}{k} \gg \binom{n}{k-1}$.

At each OR gate, the SUBSET algorithm reduces the totality $p^+ \times \binom{n}{k}$ of the input sets, and this is the only operation performed at such gates.

Observe that this is also a worst-case estimate in another sense. In Section 3.4.1, and in the SHORTCUT flow chart of Section 3.4.3, we have indicated the need to further streamline SUBSET by including the simple but very effective control loop which would prevent sets at a given input terminal from being compared with other sets at that terminal. Since this feature is not currently part of SHORTCUT, we have neglected the computational savings which it contributes.

Referring to our discussion of SUBSET in Section 3.2, we note that the processing of each OR gate thus requires $n \times p^+ \times \frac{n^k}{k!} = p^+ \times \frac{n^{k+1}}{k!}$ operations. For all the N_g^+ OR gates, a total of $N_c^+ = p^+ \times N_g^+ \times \frac{n^{k+1}}{k!}$ operations would *at most* be required.

It follows from our earlier observation in the introduction of Section 3.2 that, for other methods, at *least* $p^+ \times N_g^+ \left(\frac{n^{k+1}}{k!}\right)^2$ operations would be required, a much larger amount of work.

3.4.4.2 Computational Work at AND gates

The first step at each AND gate is the same as that executed for OR gates: collectively reduce all input sets with SUBSET. Thus $p^{\cdot} \times N_g^{\cdot} \frac{n^{k+1}}{k!}$ operations would at most be required.

For the AND-ing process itself, assume again that each set has k non-don't care entries, and that each terminal is fed $\frac{n^k}{k!}$ sets, as before.

Multiplying one terminal collection of $\frac{n^k}{k!}$ sets with another set requires k vector operations only. Multiplying with $\frac{n^k}{k!}$ other sets require $\frac{n^{k+1}}{k!}$ operations, which produces a new collection with at most $\frac{n^k}{k!}$ sets again, after set truncation with TRUNC. This truncation process is trivial and cheap, and does not increase the work significantly.

Accounting for all $p^{\cdot}$ terminals, we obtain that at most $p^{\cdot} \frac{n^{k+1}}{k!}$ AND-ing operations are required at each AND gate. For all AND gates a total of $p^{\cdot} \times N_g^{\cdot} \times \frac{n^{k+1}}{k!}$ AND-ing operations are thus required.

The resulting products must now be reduced as usual, and an additional $p^{\cdot}N_g^{\cdot}\frac{n^{k+1}}{k!}$ would be required. The total number of operations which could thus be required at AND gates is thus in the worst case

$$N_c^{\cdot} = p^{\cdot}N_g^{\cdot}\frac{n^{k+1}}{k!} + p^{\cdot}N_g^{\cdot}\frac{n^{k+1}}{k!} + p^{\cdot}N_g^{\cdot}\frac{n^{k+1}}{k!} = 3p^{\cdot}N_g^{\cdot}\frac{n^{k+1}}{k!} \quad . \tag{33}$$

3.4.4.3 Overall Complexity of SHORTCUT

Combining the total work required at OR gates and AND gates, for a fault tree with N_g^+ OR gates and $N_g^{\cdot}$ AND gates, the total number of operations required by SHORTCUT is at most

$$N_c = N_c^+ + N_c^{\cdot} = \left(p^+N_g^+ + 3p^{\cdot}N_g^{\cdot}\right)\frac{n^{k+1}}{k!} \quad . \tag{34}$$

The complexity of SHORTCUT is thus linear in the number of gates and terminals. There is no way to avoid the polynomial behavior in n, however, since all k-long sets may have to be generated.

In conclusion, a major computational improvement contributed by the SHORTCUT algorithm is the reduction of the squared term $\left(\frac{n^{k+1}}{k!}\right)^2$ to the linear term $\frac{n^{k+1}}{k!}$.

Chapter 4

CONCLUSIONS AND FUTURE WORK

SHORTCUT is a new method for finding the k-long cut sets of an arbitrary noncoherent fault tree. Given that the predominant amount of work in finding cut sets is done at the reduction stage, the method is based on a new vector algorithm—called SUBSET—for reducing sets. For typical large problems, this new algorithm is faster than existing algorithms by orders of magnitude, but does depend to some extent on the architecture of the computer used.

Two activities must yet be accomplished. First, SHORTCUT must be implemented and tested as a whole system, not just its important parts. Second, the SUBSET algorithm must be streamlined to exploit the fact that simple sets (expressions) at a given gate input need not be reduced (compared) relative to one another. For fault trees whose gates have many inputs, this will lead to further significant gains in computational efficiency.

Glossary

$\bigcup_{i=1}^{m} S_i$	Union of n sets $\{S_1, \ldots, S_m\}$.
$\bigcap_{j=1}^{m} S_j$	Intersection of m sets $\{S_1, \ldots, S_m\}$.
$\exists$	There exists.
iff	If, and only if.
$\bigvee_{i=1}^{n} A_i$	Disjunction (ORing) of n Boolean expressions $\{A_1, \ldots, A_n\}$.
$\bigwedge_{j=1}^{n} A_j$	Conjunction (ANDing) of m Boolean expressions $\{A_1, \ldots, A_n\}$.
$\wedge$ (·)	Logical AND operation.
$\vee$ (+)	Logical OR operation.
$\neg$ (–)	Logical complementation symbol.
$\bar{E}$	Boolean complement of variable E, expression E, or simple set E.
$\bar{e}_j$	Boolean complement of coordinate j entry e_j.
$\subset$	Subset symbol.
$\subsetneq$	Proper subset symbol.
$\ni$	Such that.
$A \times B$	Cartesian product of two sets A and B, $A \times B = \{(a, b): a \in A, b \in B\}$.
$\in$	Element of.
$\forall$	For all.
$\rightarrow$	Implication.

Acknowledgments

The author gratefully acknowledges the support received from the Nuclear Regulatory Commission (NRC), Division of Research. He particularly appreciates the encouragement and practical guidance received from Dr. D. M. Rasmussen, of the NRC, without whom this work might not have been accomplished.

Specific recognition is owed to C. J. Patenaude for implementing and testing some of the modules in SHORTCUT, the SUBSET algorithm in particular, and to R. M. Thatcher and J. E. Wells, of the Lawrence Livermore National Laboratory, for reviewing and criticizing our work.

Finally, the outstanding typesetting and editing work of F. McFarland was essential in producing this report.

References

[1] G. C. Corynen, "Evaluating the Response of Complex Systems to Environmental Threats: The ΣΠ Method," Lawrence Livermore National Laboratory, UCRL-53399, May 1983.

[2] E. Mendelson, "Boolean Algebra and Switching Circuits," *Shaum's outline in Mathematics*, McGraw Hill, 1970.

[3] Richard E. Barlow and Frank Proschan, *Statistical Theory of Reliability and Life Testing: Probability Models*, Holt, Rinehart, and Winston, Inc., New York, NY, 1975.

[4] Hiromitsu Kumamoto and Ernest J. Henley,"Top-down Algorithm for Obtaining Prime Implicant Sets of Non-Coherent Fault Trees," *IEEE Trans. on Reliability*, Vol. R-27, No. 4, pp. 242–249, October 1978.

[5] R. B. Worrell and D. W. Stack, *A SETS User's Manual for the Fault Tree Analyst*, NUREG/CR-0465, SAND77-2051, Unlimited Release, November 1978.

[6] R. G. Bennetts, "On the Analysis of Fault Trees," *IEEE Trans. on Reliability*, Vol. R-24, No. 3, pp. 175–185, August 1975.

[7] Bernie L. Hulme and Richard B. Worrell, "A Prime Implicant Algorithm with Factoring," *IEEE Trans. on Reliability*, Vol. R-24, No. 3, pp. 1129–1131, November 1975.

[8] R. B. Worrell, D. W. Stack, and B. L. Hulme, "Prime Implicants of Noncoherent Fault Trees," *IEEE Trans. on Reliability*, Vol. R-30, No. 2, pp. 98–100, June 1981.

[9] K. Takaragi, R. Sasaki, and S. Shingai, "An Algorithm for Obtaining Simplified Prime Implicant Sets in Fault-Tree and Event-Tree Analysis, *IEEE Trans. on Reliability*, Vol. R-32, No. 4, pp. 386–390, October 1983.

[10] R. R. Willie, *Computer-Aided Fault Tree Analysis (FTAP)*, Report Number ORC 78-14, August 1978.

[11] K. Nakashima and Y. Hattori, "An Efficient Bottom-Up Algorithm for Enumerating Minimal Cut Sets of Fault Trees," *IEEE Trans. on Reliability*, Vol. R-28, No. 5, pp. 353–357, December 1979.

[12] Mohammad Modarres, Norman C. Rasmussen, and Lothar Wolf, "Reliability Analysis of Complex Technical Systems Using the Fault Tree Modularization Technique," Massachusetts Institute of Technology, Report MITNE-228, January 1980.